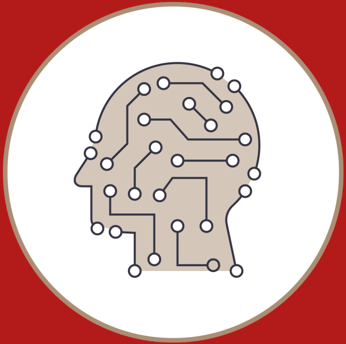
OCOM5102 – Algorithms

Introduction

Isolde Adler and Sebastian Ordyniak



Modelling in AI and Computer Science

Modelling a real-world situation is a typical task of any AI practitioner. Usually, a real-world problem that requires a solution is modelled using tools from mathematics. After that, an algorithm computes a solution in the model, and then the solution is translated back to a solution in the real world.

[1] Attributed to William of Ockham c. 1287–1347

Modelling in AI and Computer Science

Modelling a real-world situation is a typical task of any AI practitioner. Usually, a real-world problem that requires a solution is modelled using tools from mathematics. After that, an algorithm computes a solution in the model, and then the solution is translated back to a solution in the real world.

In the following chapters, we will model the **importance (quality, relevance)** of webpages by numbers and show how to compute them.

[1] Attributed to William of Ockham c. 1287–1347

Modelling in AI and Computer Science

Modelling a real-world situation is a typical task of any AI practitioner. Usually, a real-world problem that requires a solution is modelled using tools from mathematics. After that, an algorithm computes a solution in the model, and then the solution is translated back to a solution in the real world.

In the following chapters, we will model the **importance (quality, relevance)** of webpages by numbers and show how to compute them.

What aspects of a real-world situation should be modelled? Which ones left out?

- Depends on the problem at hand (modeller's choice)
- General guidance: do **not include** what you do **not need**.
  This principle is guided by **Occam's razor**[1]: *The simplest explanation is usually the right one.*

[1] Attributed to William of Ockham c. 1287–1347

Search Engine

Input: A query, consisting of one or several keywords.

Goal: A list of webpages that contain information on the keywords, sorted by relevance.

Challenges

- there are a huge number of webpages
  already in 2008: More than 1 Billion $(= 10^{12})$
- new webpages appear all the time
- many webpages are updated on a daily basis, some are deleted
- no-one knows the exact content of the entire internet
- nevertheless, search queries need to be answered in ‘real-time’

The architecture of search engines

For a rapidly changing search space of gigantic size, search queries need to be answered without noticeable reaction time.

For this, search engines use the following components:

The architecture of search engines

(1) **Web-Crawler:**
Computer programmes that search the internet, in order to identify new or changed webpages.
The information about webpages that is found by crawlers is processed and stored.

(2) **Indexing:**
The information is stored in a data structure, which supports real-time access to all webpages that contain a given keyword.

(3) **Assessing the webpages:**
The information content of the chosen webpages is assessed regarding possible keywords **and** regarding their general importance in the internet.

Search Engine

Input: A query, consisting of one or several keywords.

Goal: A list of webpages that contain information on the keywords, sorted by relevance.

Search Engine

Input: A query, consisting of one or several keywords.

Goal: A list of webpages that contain information on the keywords, sorted by relevance.

For measuring ‘relevance’, the following criteria are taken into account:

Criteria for ‘relevance’ of keywords

(1) The frequency and positioning of the search keywords on the respective webpage, and the labelling of the links to the webpage, and

(2) The fundamental importance of a webpage.

Criteria for ‘relevance’ of keywords

(1) The frequency and positioning of the search keywords on the respective webpage, and the labelling of the links to the webpage, and

(2) The fundamental importance of a webpage.

For (1) methods from the area of **information retrieval** are used.

For (2), usually, only the web-graph itself is considered.
This means that the measure for the ‘fundamental importance’ of a webpage results from the link structure of the internet only, without taking into account the textual content of a webpage.

Justification for this:

‘Fundamental importance’ of a webpage via the link structure

If a webpage i contains a link to a webpage j, then (hopefully) the following is true:

- There is a relationship between the content of both webpages, and
- The author of webpage i considers the information on webpage j to be valuable.

Measures for the ‘fundamental importance’ of a webpage

Different methods exist that yield a measure of the fundamental importance of a webpage, e. g.

- The **Page rank** method, invented by Sergei Brin and Larry Page, the founders of Google
- The **HITS** method (**H**ypertext **I**nduced **T**opic **S**earch) by Jon Kleinberg.

[2]Details of both methods can be found in the book: A. Langville and Carl D. Meyer. *Google's Pagerank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006.

Measures for the ‘fundamental importance’ of a webpage

Different methods exist that yield a measure of the fundamental importance of a webpage, e. g.

- The **Page rank** method, invented by Sergei Brin and Larry Page, the founders of Google
- The **HITS** method (**H**ypertext **I**nduced **T**opic **S**earch) by Jon Kleinberg.

Both approaches try to convert the ‘relative valuation’ between individual webpages that is manifested in the link structure into a ‘fundamental importance’ of the webpages.[2]

[2] Details of both methods can be found in the book: A. Langville and Carl D. Meyer. *Google’s Pagerank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006.

Processing a search query

. . . where a list s of search keywords is entered:

Each webpage i is assigned a value **Score(i, s)**, as a measure of relevance of webpage i regarding search query s.

As a hit list, all those webpages are returned whose score is above a certain threshold, sorted such that the webpages with highest score appear first.

The value Score(i, s)

...depends on:

- the textual content of webpage i, as well as the labels of the links pointing of i, and
- the 'fundamental importance' of webpage i.

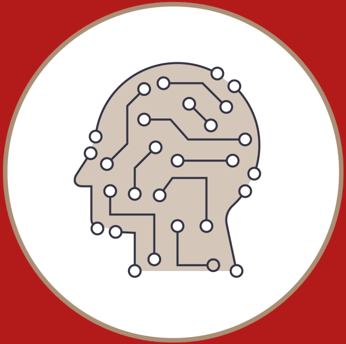
How Score(i, s) is computed exactly is an industrial secret of the operators of search engines.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Page rank

Isolde Adler and Sebastian Ordyniak



Page rank

- At the core of Google's algorithm for ranking webpages.
- Invented by Sergey Brin and Larry Page.

Modelling the webgraph

- In the following, we will use a directed graph $G = (V, E)$ to denote the webgraph.

Modelling the webgraph

- In the following, we will use a directed graph $G = (V, E)$ to denote the webgraph.
- For simplicity, we assume that the websites are numbered $1, \ldots, n$ and that $V = \{1, \ldots, n\}$.

Modelling the webgraph

- In the following, we will use a directed graph $G = (V, E)$ to denote the webgraph.
- For simplicity, we assume that the websites are numbered $1, \ldots, n$ and that $V = \{1, \ldots, n\}$.
- Every vertex of G represents a webpage, and every edge $(i, j) \in E$ models a link from webpage i to webpage j.

Modelling the webgraph

- In the following, we will use a directed graph $G = (V, E)$ to denote the webgraph.
- For simplicity, we assume that the websites are numbered $1, \dots, n$ and that $V = \{1, \dots, n\}$.
- Every vertex of G represents a webpage, and every edge $(i, j) \in E$ models a link from webpage i to webpage j.
- For a webpage $j \in V$ we write $N^-(j)$ to denote the set of all webpages that contain a link to j, i. e.

$$N^-(j) := \{i \in V \mid (i, j) \in E\}.$$

Modelling the webgraph

- In the following, we will use a directed graph $G = (V, E)$ to denote the webgraph.
- For simplicity, we assume that the websites are numbered $1, \ldots, n$ and that $V = \{1, \ldots, n\}$.
- Every vertex of G represents a webpage, and every edge $(i, j) \in E$ models a link from webpage i to webpage j.
- For a webpage $j \in V$ we write $N^-(j)$ to denote the set of all webpages that contain a link to j, i. e.

$$N^-(j) := \{i \in V \mid (i, j) \in E\}.$$

- We call the elements of $N^-(j)$ **predecessors** of j.

Page rank: the idea

We will model the **fundamental importance** of webpage i by a number PR_i, the **page rank** of i.

Page rank: the idea

We will model the **fundamental importance** of webpage i by a number PR_i, the **page rank** of i.

- PR_i is meant to represent the quality (in the sense of **reputation**) of webpage i.

Page rank: the idea

We will model the **fundamental importance** of webpage i by a number PR_i, the **page rank** of i.

- PR_i is meant to represent the quality (in the sense of **reputation**) of webpage i.
- The larger PR_i is, the bigger the reputation of i.

Page rank: the idea

We will model the **fundamental importance** of webpage i by a number PR_i, the **page rank** of i.

- PR_i is meant to represent the quality (in the sense of **reputation**) of webpage i.
- The larger PR_i is, the bigger the reputation of i.
- PR_j of webpage j is high, if many pages i with high page rank PR_i point to j.

Page rank: the idea

We will model the **fundamental importance** of webpage i by a number PR_i, the **page rank** of i.

- PR_i is meant to represent the quality (in the sense of **reputation**) of webpage i.
- The larger PR_i is, the bigger the reputation of i.
- PR_j of webpage j is high, if many pages i with high page rank PR_i point to j.
- The values PR_i associated to all webpages $i \in V$ are chosen such that: Webpage i with out_i outgoing links passes the fraction

$$\frac{\mathrm{PR}_i}{\mathrm{out}_i}$$

  of its page rank on to each webpage j with $(i, j) \in E$.

Page rank: the idea

We will model the **fundamental importance** of webpage i by a number PR_i, the **page rank** of i.

- PR_i is meant to represent the quality (in the sense of **reputation**) of webpage i.
- The larger PR_i is, the bigger the reputation of i.
- PR_j of webpage j is high, if many pages i with high page rank PR_i point to j.
- The values PR_i associated to all webpages $i \in V$ are chosen such that: Webpage i with out_i outgoing links passes the fraction
$$\frac{\mathrm{PR}_i}{\mathrm{out}_i}$$
of its page rank on to each webpage j with $(i, j) \in E$.
- With this, for each $j \in V$ with $N^-(j) \neq \emptyset$ we would obtain
$$\mathrm{PR}_j = \sum_{i \in N^-(j)} \frac{\mathrm{PR}_i}{\mathrm{out}_i}.$$

Issue 1: sinks

With the proposed definition, vertices of out-degree 0 cause a problem: they do not pass on their page rank to other vertices, and they can lead to values PR_i that do not make sense as a measure of importance of a webpage.

Issue 1: sinks

With the proposed definition, vertices of out-degree 0 cause a problem: they do not pass on their page rank to other vertices, and they can lead to values PR_i that do not make sense as a measure of importance of a webpage.

Definition: A vertex of out-degree 0 is also called a **sink**.

Sinks: example

Let $G = (V, E)$ be the following graph:

Sinks: example

Let $G = (V, E)$ be the following graph:

The only values $\mathrm{PR}_1, \mathrm{PR}_2, \mathrm{PR}_3, \mathrm{PR}_4 \in \mathbb{R}$ that satisfy

$$\mathrm{PR}_j = \sum_{i \in N_G^-(j)} \frac{\mathrm{PR}_i}{\mathrm{out}_i} \quad \text{for all } j \in \{1, 2, 3, 4\}$$

are $\mathrm{PR}_1 = \mathrm{PR}_2 = \mathrm{PR}_3 = \mathrm{PR}_4 = 0$.

Removing sinks

In order to determine the page rank, we only consider **graphs without sinks**, i. e. directed graphs, where every vertex has out-degree ≥ 1.

Of course there is no guarantee that the webgraph contains no sinks. Brin and Page propose two ways of transforming the web graph into a graph without sinks:

Option 1:
For every sink i, add additional edges (i, j) to **all** $j \in V$.

Option 2:
Delete all sinks and repeat this until a graph without sinks is obtained.

General assumption

From now on, we will assume that one of the two transformations was done and that the web graph can be represented by a finite directed graph $G = (V, E)$ without any sinks.

Issue 2: strongly connected sets

Another problem is caused by sets of vertices, that may be connected with each other, but do not contain an edge to any vertex of G outside the set.

Similar to sinks, such vertex sets can lead to values PR_i that are not a reasonable measure for the 'general importance' of webpages.

Strongly connected sets with no out-edge: example

Let $G = (V, E)$ be the following graph:

Strongly connected sets with no out-edge: example

Let $G = (V, E)$ be the following graph:

Here all of $\mathrm{PR}_1, \ldots, \mathrm{PR}_5$ satisfy:

$$\mathrm{PR}_j = \sum_{i \in N_G^-(j)} \frac{\mathrm{PR}_i}{\mathrm{out}_i} \quad \text{for all } j \in \{1, \ldots, 5\}$$

if and only if $\mathrm{PR}_1 = \mathrm{PR}_2 = \mathrm{PR}_3 = 0$ and $\mathrm{PR}_4 = \mathrm{PR}_5$.

Strongly connected sets with no out-edge: example

Let $G = (V, E)$ be the following graph:

Here all of $\mathrm{PR}_1, \ldots, \mathrm{PR}_5$ satisfy:

$$\mathrm{PR}_j = \sum_{i \in N_G^-(j)} \frac{\mathrm{PR}_i}{\mathrm{out}_i} \quad \text{for all } j \in \{1, \ldots, 5\}$$

if and only if $\mathrm{PR}_1 = \mathrm{PR}_2 = \mathrm{PR}_3 = 0$ and $\mathrm{PR}_4 = \mathrm{PR}_5$.

Note: In particular, we can choose any number for $\mathrm{PR}_4 = \mathrm{PR}_5$!

Damping factor

In order to avoid this problem, we introduce a so-called **damping factor** d with $0 \leq d \leq 1$, that damps the proportion of PR_i that is passed on to j with $(i, j) \in E$ by a factor d.

This is made precise in the following definition.

Brin and Page recommend choosing $d = 0.85 = \frac{17}{20}$.

Page rank property: definition

Let $d \in \mathbb{R}$ with $0 \leq d \leq 1$ be the **damping factor**. Let $G = (V, E)$ be a directed graph without sinks, and let $n := |V| \in \mathbb{N}_{>0}$ and $V = \{1, \ldots, n\}$.

Page rank property: definition

Let $d \in \mathbb{R}$ with $0 \leq d \leq 1$ be the **damping factor**. Let $G = (V, E)$ be a directed graph without sinks, and let $n := |V| \in \mathbb{N}_{>0}$ and $V = \{1, \dots, n\}$.

Recall that $N^-(i) := \{j \in V : (j, i) \in E\}$, for $i \in V$.

A tuple $\mathrm{PR} = (\mathrm{PR}_1, \dots, \mathrm{PR}_n) \in \mathbb{R}^n$ has the **page rank property with respect to d**, if ever $j \in V$ satisfies

$$\mathrm{PR}_j = \frac{1-d}{n} + d \cdot \sum_{i \in N^-(j)} \frac{\mathrm{PR}_i}{\mathrm{out}_i}.$$

Page rank property: special cases

- For $d = 1$ we obtain the equation

$$\mathrm{PR}_j = \sum_{i \in N_G^-(j)} \frac{\mathrm{PR}_i}{\mathrm{out}_i}.$$

Page rank property: special cases

- For $d = 1$ we obtain the equation

$$\mathrm{PR}_j = \sum_{i \in N_G^-(j)} \frac{\mathrm{PR}_i}{\mathrm{out}_i}.$$

- For $d = 0$ we obtain

$$\mathrm{PR}_j = \frac{1}{n}$$

  for all $j \in V$.

Page rank property: example

Let $d := 1/2$ and let $G = (V, E)$ be the following graph:

Task: Find a tuple $\mathrm{PR} = (\mathrm{PR}_1, \mathrm{PR}_2, \mathrm{PR}_3) \in \mathbb{R}^3$ that has the page rank property with respect to d.

Page rank property: example

We are looking for a tuple $\mathrm{PR} = (\mathrm{PR}_1, \mathrm{PR}_2, \mathrm{PR}_3) \in \mathbb{R}^3$ that has the page rank property with respect to d, i. e. we must have:

1. $\mathrm{PR}_1 = 1/(2 \cdot 3) + 1/2 \cdot \mathrm{PR}_3 / 1$
2. $\mathrm{PR}_2 = 1/(2 \cdot 3) + 1/2 \cdot \mathrm{PR}_3 / 2$
3. $\mathrm{PR}_3 = 1/(2 \cdot 3) + 1/2 \cdot (\mathrm{PR}_1 / 2 + \mathrm{PR}_2 / 1)$

Page rank property: example

We are looking for a tuple $\mathrm{PR} = (\mathrm{PR}_1, \mathrm{PR}_2, \mathrm{PR}_3) \in \mathbb{R}^3$ that has the page rank property with respect to d, i. e. we must have:

1. $\mathrm{PR}_1 = 1/(2 \cdot 3) + 1/2 \cdot \mathrm{PR}_3 /1$
2. $\mathrm{PR}_2 = 1/(2 \cdot 3) + 1/2 \cdot \mathrm{PR}_3 /2$
3. $\mathrm{PR}_3 = 1/(2 \cdot 3) + 1/2 \cdot (\mathrm{PR}_1 /2 + \mathrm{PR}_2 /1)$

Hence we can find the values $\mathrm{PR}_1, \mathrm{PR}_2,$ and PR_3 by solving the following system of linear equations:

1. $1 \cdot \mathrm{PR}_1 - 1/2 \cdot \mathrm{PR}_3 = 1/6$
2. $-1/4 \cdot \mathrm{PR}_1 + 1 \cdot \mathrm{PR}_2 = 1/6$
3. $-1/4 \cdot \mathrm{PR}_1 - 1/2 \cdot \mathrm{PR}_2 = 1 \cdot \mathrm{PR}_3 = 1/6$

Page rank property: example

We are looking for a tuple $\mathrm{PR} = (\mathrm{PR}_1, \mathrm{PR}_2, \mathrm{PR}_3) \in \mathbb{R}^3$ that has the page rank property with respect to d, i. e. we must have:

1. $\mathrm{PR}_1 = 1/(2 \cdot 3) + 1/2 \cdot \mathrm{PR}_3 /1$
2. $\mathrm{PR}_2 = 1/(2 \cdot 3) + 1/2 \cdot \mathrm{PR}_3 /2$
3. $\mathrm{PR}_3 = 1/(2 \cdot 3) + 1/2 \cdot (\mathrm{PR}_1 /2 + \mathrm{PR}_2 /1)$

Hence we can find the values $\mathrm{PR}_1, \mathrm{PR}_2,$ and PR_3 by solving the following system of linear equations:

1. $1 \cdot \mathrm{PR}_1 - 1/2 \cdot \mathrm{PR}_3 = 1/6$
2. $-1/4 \cdot \mathrm{PR}_1 + 1 \cdot \mathrm{PR}_2 = 1/6$
3. $-1/4 \cdot \mathrm{PR}_1 - 1/2 \cdot \mathrm{PR}_2 = 1 \cdot \mathrm{PR}_3 = 1/6$

Solving this, e. g. using **Gauss elimination**, yields

$$\mathrm{PR}_1 = 14/39, \mathrm{PR}_2 = 10/39, \mathrm{PR}_3 = 15/39.$$

Computing the page rank

For the actual web graph and a suitable damping factor d we also obtain a system of linear equations, similar to the previous example.

In order to compute the page rank of all webpages, we only need to solve this system of linear equations.

Problem 1

To begin with, it is unclear whether the system of linear equations has a solution at all, and if it does, it is unclear whether the solution is unique.

Problem 2

The system of linear equations has n unknowns, where n is the number of webpages in the internet – this is a huge number!
Hence we need a very efficient method to solve the system of equations.

Solution

Use results and methods from the theory of **Markov chains**.

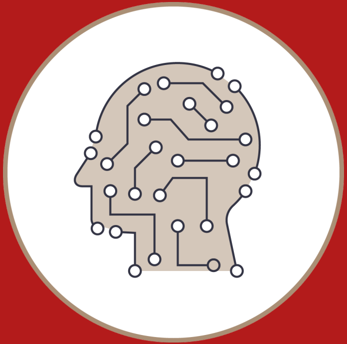
For explaining the connection between Markov chains and page rank, the view of page rank via a **random surfer** is helpful. We will discuss this in the next presentation.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

The random surfer

Isolde Adler and Sebastian Ordyniak



The random surfer

- Let $G = (V, E)$ with $V = \{1, \ldots, n\}$ and $n \in \mathbb{N}_{>0}$ be a directed graph without sinks, representing the web graph.
- Let $d \in \mathbb{R}$ with $0 \leq d \leq 1$.

Consider a **random surfer**, who starts on an arbitrary webpage and follows arbitrary links, without paying attention to the contents of the webpages.

The random surfer

If the random surfer is at a webpage i,

- With probability d the surfer chooses some link, that starts on page i.

  Here each of the out_i links is chosen with the same probability $\frac{d}{\mathrm{out}_i}$.

- With probability $1 - d$ the surfer chooses a random webpage in V.

  Here each of the n webpages is chosen with the same probability $\frac{1-d}{n}$.

The random surfer

Hence for all $i, j \in V$, the number

$$\boldsymbol{p_{i,j}} := \begin{cases} \frac{1-d}{n} + \frac{d}{\mathrm{out}_i}, & \text{if } (i,j) \in E \\ \frac{1-d}{n}, & \text{if } (i,j) \notin E \end{cases}$$

is the **probability** for the random surfer to move **from page i to page j** in one step.

These probabilities of the random surfer moving from vertex to vertex can be compactly represented by a matrix as follows:

Page rank matrix: definition

Let $d \in \mathbb{R}$ with $0 \leq d \leq 1$, let $n \in \mathbb{N}_{>0}$ and let $G = (V, E)$ with $V = \{1, \ldots, n\}$ be a directed graph without sinks.

The **page rank matrix** is the $(n \times n)$-matrix

$$\boldsymbol{P(G,d)} := \begin{pmatrix} p_{1,1} & \cdots & p_{1,j} & \cdots & p_{1,n} \\ \vdots & & \vdots & & \vdots \\ p_{i,1} & \cdots & p_{i,j} & \cdots & p_{i,n} \\ \vdots & & \vdots & & \vdots \\ p_{n,1} & \cdots & p_{n,j} & \cdots & p_{n,n} \end{pmatrix},$$

where for every $i, j \in V$ the entry in row i and column j is the value $p_{i,j}$ defined on the previous page. We also use $(p_{i,j})_{i,j=1,\ldots,n}$ to denote the matrix $P(G, d)$.

Page rank matrix: example

Let $d = 1/2$ and let $G = (V, E)$ be the graph from the previous example (illustrating the page rank property):

Task: Find $p_{1,1}, p_{1,2}$ and $p_{2,3}$ and $P(G, d)$.

Page rank matrix: example

Let $d = 1/2$ and let $G = (V, E)$ be the graph from the previous example (illustrating the page rank property):

Task: Find $p_{1,1}, p_{1,2}$ and $p_{2,3}$ and $P(G, d)$.

Using the definition of $p_{i,j}$ we obtain $p_{1,1} = 1/6$, $p_{1,2} = 1/6 + 1/4 = 5/12$, and $p_{2,3} = 1/6 + 1/2 = 2/3$.

Page rank matrix: example

Let $d = 1/2$ and let $G = (V, E)$ be the graph from the previous example (illustrating the page rank property):

Task: Find $p_{1,1}, p_{1,2}$ and $p_{2,3}$ and $P(G, d)$.

Using the definition of $p_{i,j}$ we obtain $p_{1,1} = 1/6$, $p_{1,2} = 1/6 + 1/4 = 5/12$, and $p_{2,3} = 1/6 + 1/2 = 2/3$.

Altogether, we get

$$P(G, d) = \begin{pmatrix} 1/6 & 5/12 & 5/12 \\ 1/6 & 1/6 & 2/3 \\ 2/3 & 1/6 & 1/6 \end{pmatrix}.$$

For describing the connection between page rank and random surfer, we need the notion of **vector matrix product**.

Vector matrix product: definition

Let $n \in \mathbb{N}_{>0}$, and for all $i, j \in \{1, \ldots, n\}$ let $p_{i,j}$ be a real number. Let $P := (p_{i,j})_{i,j=1,\ldots,n}$ be the $(n \times n)$-matrix with entry $p_{i,j}$ in row i and column j (for all $i, j \in \{1, \ldots, n\}$). For a tuple $X = (X_1, \ldots, X_n)$ of n real numbers, the **vector matrix product**

$$\boldsymbol{X \cdot P}$$

is the tuple $\boldsymbol{Y = (Y_1, \ldots, Y_n)} \in \mathbb{R}^{\boldsymbol{n}}$, where for every $j \in \{1, \ldots, n\}$ we have

$$\boldsymbol{Y_j} := \sum_{i=1}^{n} X_i \cdot p_{i,j}.$$

Vector matrix product: example

Let $P := P(G, d)$ be the matrix from the previous example and let $X := (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
Then:

Vector matrix product: example

Let $P := P(G, d)$ be the matrix from the previous example and let $X := (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
Then:

$$
\begin{aligned}
X \cdot P &= (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}) \cdot \begin{pmatrix} 1/6 & 5/12 & 5/12 \\ 1/6 & 1/6 & 2/3 \\ 2/3 & 1/6 & 1/6 \end{pmatrix} \\
&= (\frac{1}{3} \cdot \frac{1}{6} + \frac{1}{3} \cdot \frac{1}{6} + \frac{1}{3} \cdot \frac{2}{3}, \quad \frac{1}{3} \cdot \frac{5}{12} + \frac{1}{3} \cdot \frac{1}{6} + \frac{1}{3} \cdot \frac{1}{6}, \quad \frac{1}{3} \cdot \frac{5}{12} + \frac{1}{3} \cdot \frac{2}{3} + \frac{1}{3} \cdot \frac{1}{6}) \\
&= (\frac{1}{3}, \frac{1}{4}, \frac{5}{12}).
\end{aligned}
$$

The following theorem shows the connection between

- The random surfer,
- The page rank matrix, and
- Tuples with the page rank property.

Page rank, random surfer & page rank property: theorem

Theorem:
Let $d \in \mathbb{R}$ with $0 \leq d < 1$, let $n \in \mathbb{N}_{>0}$ and let $G = (V, E)$ be a directed graph with $V = \{1, \ldots, n\}$ without sinks. Then:

(a) If $\mathrm{PR} = (\mathrm{PR}_1, ..., \mathrm{PR}_n) \in \mathbb{R}^n$ is a tuple that has the page rank property with respect to d, then $\sum_{i=1}^n \mathrm{PR}_i = 1$.

(b) Every tuple $X = (X_1, \ldots, X_n) \in \mathbb{R}^n$ with $\sum_{i=1}^n X_i = 1$ satisfies

$$X \text{ has the page rank property with respect to } d \iff X \cdot P(G, d) = X.$$

Proof: Cf. the next video.

Note: For the proof of (a) it will be important that $d \neq 1$, and that G has no sinks.

Notation

A vector $X = (X_1, \ldots, X_n)$ is called **left eigenvector with eigenvalue 1** of the $(n \times n)$-matrix P, if

$$\boldsymbol{X \cdot P = X}$$

and $X \neq (0, \ldots, 0)$.

Hence the previous theorem says that a tuple $\mathrm{PR} = (\mathrm{PR}_1, \dots, \mathrm{PR}_n) \in \mathbb{R}^n$ has the page rank property with respect to d, if and only if it is a left eigenvector with eigenvalue 1 of the matrix $P(G, d)$ satisfying $\sum_{i=1}^{n} \mathrm{PR}_i = 1$.

Hence the previous theorem says that a tuple $\mathrm{PR} = (\mathrm{PR}_1, \dots, \mathrm{PR}_n) \in \mathbb{R}^n$ has the page rank property with respect to d, if and only if it is a left eigenvector with eigenvalue 1 of the matrix $P(G, d)$ satisfying $\sum_{i=1}^n \mathrm{PR}_i = 1$.

This view on the page rank, as well as the theory of Markov chains, which we will explain in the next section, will help us answer Problems (1) and (2).

Hence the previous theorem says that a tuple $\mathrm{PR} = (\mathrm{PR}_1, \ldots, \mathrm{PR}_n) \in \mathbb{R}^n$ has the page rank property with respect to d, if and only if it is a left eigenvector with eigenvalue 1 of the matrix $P(G, d)$ satisfying $\sum_{i=1}^n \mathrm{PR}_i = 1$.

This view on the page rank, as well as the theory of Markov chains, which we will explain in the next section, will help us answer Problems (1) and (2).

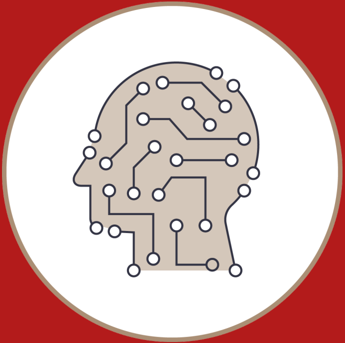
Reminder
Problem (1): Existence and uniqueness of tuples with the page rank property.
Problem (2): Efficient procedures for the computation of such tuples.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Markov chains

Isolde Adler and Sebastian Ordyniak



Markov chains

Named after the Russian mathematician Andrei A. Markov, 1856 – 1922.

Other spellings: Markow, Markoff.

Markov chain: definition

A **(homogeneous) Markov chain with transition matrix P** is described by an $(n \times n)$-matrix

$$P = (p_{i,j})_{i,j=1,\ldots,n}$$

with $n \in \mathbb{N}_{>0}$, such that:

(1) $p_{i,j} \geq 0$ for all $i, j \in \{1, \ldots, n\}$, and

(2) every row $i \in \{1, \ldots, n\}$ satisfies

$$\sum_{j=1}^{n} p_{i,j} = 1.$$

A matrix $P = (p_{i,j})_{i,j=1,\ldots,n}$ with properties (1) and (2), is called a **stochastic matrix**.

Markov chain: definition continued

The **graph corresponding to P** is the directed graph with vertex set $V = \{1, \ldots, n\}$, such that for all $i, j \in \{1, \ldots, n\}$ we have

$$(i, j) \in E \iff p_{i,j} > 0.$$

Entry $p_{i,j}$ in row i and column j of P represents the probability that a random surfer in graph G moves from vertex i to vertex j in one step.

Markov chain: example

Let $G = (V, E)$ be an arbitrary directed graph with vertex set $V = \{1, ..., n\}$ (for some $n \in \mathbb{N}_{>0}$) and without sinks.
Let $d \in \mathbb{R}$ with $0 \leq d < 1$, and let $P = P(G, d)$ be the corresponding page rank matrix.

Markov chain: example

Let $G = (V, E)$ be an arbitrary directed graph with vertex set $V = \{1, ..., n\}$ (for some $n \in \mathbb{N}_{>0}$) and without sinks.
Let $d \in \mathbb{R}$ with $0 \leq d < 1$, and let $P = P(G, d)$ be the corresponding page rank matrix.
By definition of P, we have that $p_{i.j} > 0$ for all $i, j \in \{1, \ldots, n\}$. Moreover, every row $i \in \{1, \ldots, n\}$ satisfies

$$\sum_{j=1}^{n} p_{i,j} = \sum_{j=1}^{n} \frac{1-d}{n} + \sum_{j:(i,j)\in E} \frac{d}{\mathrm{out}_i} \overset{G \text{ has no sink}}{=} (1-d) + \mathrm{out}_i \cdot \frac{d}{\mathrm{out}_i} = 1.$$

Markov chain: example

Let $G = (V, E)$ be an arbitrary directed graph with vertex set $V = \{1, ..., n\}$ (for some $n \in \mathbb{N}_{>0}$) and without sinks.
Let $d \in \mathbb{R}$ with $0 \leq d < 1$, and let $P = P(G, d)$ be the corresponding page rank matrix.
By definition of P, we have that $p_{i.j} > 0$ for all $i, j \in \{1, \ldots, n\}$. Moreover, every row $i \in \{1, \ldots, n\}$ satisfies

$$\sum_{j=1}^{n} p_{i,j} = \sum_{j=1}^{n} \frac{1-d}{n} + \sum_{j:(i,j)\in E} \frac{d}{\mathrm{out}_i} \overset{G \text{ has no sink}}{=} (1-d) + \mathrm{out}_i \cdot \frac{d}{\mathrm{out}_i} = 1.$$

Hence:

- P is a stochastic matrix that describes a Markov chain.

Markov chain: example

Let $G = (V, E)$ be an arbitrary directed graph with vertex set $V = \{1, ..., n\}$ (for some $n \in \mathbb{N}_{>0}$) and without sinks.
Let $d \in \mathbb{R}$ with $0 \leq d < 1$, and let $P = P(G, d)$ be the corresponding page rank matrix.
By definition of P, we have that $p_{i.j} > 0$ for all $i, j \in \{1, \ldots, n\}$. Moreover, every row $i \in \{1, \ldots, n\}$ satisfies

$$\sum_{j=1}^{n} p_{i,j} = \sum_{j=1}^{n} \frac{1-d}{n} + \sum_{j:(i,j)\in E} \frac{d}{\mathrm{out}_i} \overset{G \text{ has no sink}}{=} (1-d) + \mathrm{out}_i \cdot \frac{d}{\mathrm{out}_i} = 1.$$

Hence:

- P is a stochastic matrix that describes a Markov chain.
- For every $i, j \in \{1, \ldots, n\}$ the value $p_{i,j}$ is the probability that the random surfer moves from i to j in one step.

Markov chain: example continued

Since $p_{i,j} > 0$, the graph corresponding to P is the **complete directed graph on n vertices**, i. e. the graph with vertex set $V = \{1, \ldots, n\}$ and edge set $V \times V$.

Markov chain: example continued

Since $p_{i,j} > 0$, the graph corresponding to P is the **complete directed graph on n vertices**, i. e. the graph with vertex set $V = \{1, \ldots, n\}$ and edge set $V \times V$.

Notation: We use $\overrightarrow{\boldsymbol{K}}_{\boldsymbol{n}}$ to denote the complete directed graph on n vertices.

Vertex set: $V = \{1, \ldots, n\}$
Edge set: $E = V \times V$

Stochastic matrices

Markov chains and stochastic matrices are well understood. In particular, the following theorem is known.

Stochastic matrices: theorem

Let $n \in \mathbb{N}_{>0}$ and let $P = (p_{i,j})_{i,j=1,\ldots,n}$ be a stochastic matrix, such that for all $i, j \in \{1, \ldots, n\}$: $p_{i,j} > 0$.

Then there is **exactly one** tuple $X = (X_1, \ldots, X_n) \in \mathbb{R}^n$ with $\sum_{i=1}^{n} X_i = 1$, that is a left eigenvector with eigenvalue 1 of P.
This tuple has the property that for every $i \in \{1, \ldots, n\}$ we have $X_i > 0$.

Stochastic matrices: theorem

Let $n \in \mathbb{N}_{>0}$ and let $P = (p_{i,j})_{i,j=1,\ldots,n}$ be a stochastic matrix, such that for all $i, j \in \{1, \ldots, n\}$: $p_{i,j} > 0$.

Then there is **exactly one** tuple $X = (X_1, \ldots, X_n) \in \mathbb{R}^n$ with $\sum_{i=1}^{n} X_i = 1$, that is a left eigenvector with eigenvalue 1 of P.
This tuple has the property that for every $i \in \{1, \ldots, n\}$ we have $X_i > 0$.

Proving this theorem is beyond the scope of this module.

We will use it to obtain a solution of **Problem (1)**.

Stochastic matrices: corollary (solution of Problem (1))

Let $G = (V, E)$ be a directed graph with $V = \{1, \dots, n\}$ for $n \in \mathbb{N}_{>0}$ without sinks. Let $d \in \mathbb{R}$ be a damping factor with $0 \leq d < 1$.

Then there is exactly one tuple $\mathrm{PR} = (\mathrm{PR}_1, \dots, \mathrm{PR}_n) \in \mathbb{R}^n$, that has the page rank property with respect to d.

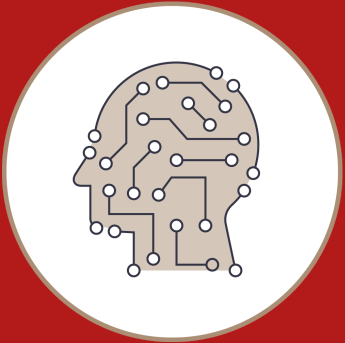
This tuple satisfies $\mathrm{PR}_i > 0$ for all $i \in \{1, \dots, n\}$ and $\sum_{i=1}^{n} \mathrm{PR}_i = 1$.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Efficient computation of the page rank

Isolde Adler and Sebastian Ordyniak



Efficient computation of the page rank

For this, we consider the way the random surfer moves on the web graph in greater detail.

The following notion is very useful:

Distribution: definition

Let $n \in \mathbb{N}_{>0}$.
A **distribution** on $V = \{1, \ldots, n\}$ is a tuple $X = (X_1, \ldots, X_n) \in \mathbb{R}^n$ satisfying:

(1) For all $i \in \{1, \ldots, n\}$ we have $X_i \geq 0$ and

(2) $\sum_{i=1}^{n} X_i = 1$.

Distribution: definition

Let $n \in \mathbb{N}_{>0}$.
A **distribution** on $V = \{1, \ldots, n\}$ is a tuple $X = (X_1, \ldots, X_n) \in \mathbb{R}^n$ satisfying:

(1) For all $i \in \{1, \ldots, n\}$ we have $X_i \geq 0$ and

(2) $\sum_{i=1}^{n} X_i = 1$.

X_i will represent the probability for the random surfer to be on vertex i.

Distribution: observation

Let $n \in \mathbb{N}_{>0}$ and let $P = (p_{i,j})_{i,j=1,\dots,n}$ be a stochastic matrix.
Let $X = (X_1, \dots, X_n)$ be a distribution on $V := \{1, \dots, n\}$. For all $i \in \{1, \dots, n\}$ let X_i be the probability that the random surfer starts on i.

Distribution: observation

Let $n \in \mathbb{N}_{>0}$ and let $P = (p_{i,j})_{i,j=1,\dots,n}$ be a stochastic matrix.
Let $X = (X_1, \dots, X_n)$ be a distribution on $V := \{1, \dots, n\}$. For all $i \in \{1, \dots, n\}$ let X_i be the probability that the random surfer starts on i. For all $j \in \{1, \dots, n\}$ we have:
The probability that the random surfer is on j after one step is:

$$Y_j := \sum_{i=1}^{n} X_i \cdot p_{i,j}.$$

Distribution: observation

Let $n \in \mathbb{N}_{>0}$ and let $P = (p_{i,j})_{i,j=1,\dots,n}$ be a stochastic matrix.
Let $X = (X_1, \dots, X_n)$ be a distribution on $V := \{1, \dots, n\}$. For all $i \in \{1, \dots, n\}$ let X_i be the probability that the random surfer starts on i. For all $j \in \{1, \dots, n\}$ we have:
The probability that the random surfer is on j after one step is:

$$Y_j := \sum_{i=1}^{n} X_i \cdot p_{i,j}.$$

Inductively, for each $k \in \mathbb{N}$ we can specify a distribution $X^{(k)} = (X_1^{(k)}, \dots, X_n^{(k)})$, such that for every $j \in V$ the probability that the random surfer is on webpage j after k steps is given by $X_j^{(k)}$. For this we let:

$$X^{(0)} := X \text{ and } X^{(k+1)} := X^{(k)} \cdot P \text{ for all } k \in \mathbb{N}.$$

Distribution: observation

Let $n \in \mathbb{N}_{>0}$ and let $P = (p_{i,j})_{i,j=1,\dots,n}$ be a stochastic matrix.
Let $X = (X_1, \dots, X_n)$ be a distribution on $V := \{1, \dots, n\}$. For all $i \in \{1, \dots, n\}$ let X_i be the probability that the random surfer starts on i. For all $j \in \{1, \dots, n\}$ we have:
The probability that the random surfer is on j after one step is:

$$Y_j := \sum_{i=1}^{n} X_i \cdot p_{i,j}.$$

Inductively, for each $k \in \mathbb{N}$ we can specify a distribution $X^{(k)} = (X_1^{(k)}, \dots, X_n^{(k)})$, such that for every $j \in V$ the probability that the random surfer is on webpage j after k steps is given by $X_j^{(k)}$. For this we let:

$$X^{(0)} := X \text{ and } X^{(k+1)} := X^{(k)} \cdot P \text{ for all } k \in \mathbb{N}.$$

By induction (on k) it can be shown that all $k \in \mathbb{N}_{>0}$ satisfy $X^{(k+1)} = X \cdot P^k$.
This notation uses matrix products, defined as follows:

Matrix product: definition

Let $n \in \mathbb{N}_{>0}$, and for all $i, j \in \{1, \ldots, n\}$ let $a_{i,j} \in \mathbb{R}$ and $b_{i,j} \in \mathbb{R}$. Consider the two $(n \times n)$ matrices

$$A = (a_{i,j})_{i,j=1,\ldots,n} \text{ and } B = (b_{i,j})_{i,j=1,\ldots,n}.$$

Matrix product: definition

Let $n \in \mathbb{N}_{>0}$, and for all $i, j \in \{1, \ldots, n\}$ let $a_{i,j} \in \mathbb{R}$ and $b_{i,j} \in \mathbb{R}$. Consider the two $(n \times n)$ matrices

$$A = (a_{i,j})_{i,j=1,\ldots,n} \text{ and } B = (b_{i,j})_{i,j=1,\ldots,n}.$$

(a) The **product $A \cdot B$** is the $(n \times n)$-matrix $C = (c_{i,j})_{i,j=1,\ldots,n}$, such that for all $i, j \in \{1, \ldots, n\}$, the entry in row i and column j is

$$c_{i,j} := \sum_{\ell=1}^{n} a_{i,\ell} \cdot b_{\ell,j}.$$

Matrix product: definition

Let $n \in \mathbb{N}_{>0}$, and for all $i, j \in \{1, \ldots, n\}$ let $a_{i,j} \in \mathbb{R}$ and $b_{i,j} \in \mathbb{R}$. Consider the two $(n \times n)$ matrices

$$A = (a_{i,j})_{i,j=1,\ldots,n} \text{ and } B = (b_{i,j})_{i,j=1,\ldots,n}.$$

(a) The **product** $\boldsymbol{A \cdot B}$ is the $(n \times n)$-matrix $C = (c_{i,j})_{i,j=1,\ldots,n}$, such that for all $i, j \in \{1, \ldots, n\}$, the entry in row i and column j is

$$c_{i,j} := \sum_{\ell=1}^{n} a_{i,\ell} \cdot b_{\ell,j}.$$

(b) For every $k \in \mathbb{N}_{>0}$ the $(n \times n)$-matrix A^k is recursively defined as follows.

$$\boldsymbol{A^1} := A \text{ and } \boldsymbol{A^{k+1}} := A \cdot A^k \text{ for all } k \in \mathbb{N}.$$

Matrix product: definition

Let $n \in \mathbb{N}_{>0}$, and for all $i, j \in \{1, \dots, n\}$ let $a_{i,j} \in \mathbb{R}$ and $b_{i,j} \in \mathbb{R}$. Consider the two $(n \times n)$ matrices

$$A = (a_{i,j})_{i,j=1,\dots,n} \text{ and } B = (b_{i,j})_{i,j=1,\dots,n}.$$

(a) The **product $A \cdot B$** is the $(n \times n)$-matrix $C = (c_{i,j})_{i,j=1,\dots,n}$, such that for all $i, j \in \{1, \dots, n\}$, the entry in row i and column j is

$$c_{i,j} := \sum_{\ell=1}^{n} a_{i,\ell} \cdot b_{\ell,j}.$$

(b) For every $k \in \mathbb{N}_{>0}$ the $(n \times n)$-matrix A^k is recursively defined as follows.

$$\boldsymbol{A^1} := A \text{ and } \boldsymbol{A^{k+1}} := A \cdot A^k \text{ for all } k \in \mathbb{N}.$$

For $i, j \in \{1, \dots, n\}$ we write $\boldsymbol{(A^k)_{i,j}}$, to denote the entry in row i and column j of the matrix A^k.

Observation

Let $n \in N_{>0}$ and let $P = (p_{i,j})_{i,j=1,\dots,n}$ be a stochastic matrix describing a Markov chain.

For every $k \in \mathbb{N}_{>0}$ and all $i, j \in \{1, \dots, n\}$ the number $(P^k)_{i,j}$ is the **probability that**, on the graph corresponding to P, **the random surfer moves from vertex i to vertex j in exactly k steps**.

For computing the page rank efficiently, we make use of the fact, that the Markov chain described by the page rank matrix $P(G, d)$ (for $0 \leq d < 1$) has the following property.

Ergodic Markov chain: definition

Let $n \in \mathbb{N}_{>0}$ and let $P = (p_{i,j})_{i,j=\{1,\ldots,n\}}$ be a stochastic matrix. The Markov chain describend by P is called **ergodic**, if all $i, i' \in \{1, \ldots, n\}$ and all $j \in \{1, \ldots, n\}$ satisfy the following: The limits

$$\lim_{k\to\infty} (P^k)_{i,j} \quad \text{and} \quad \lim_{k\to\infty} (P^k)_{i',j}$$

exist and they satisfy

$$\lim_{k\to\infty} (P^k)_{i,j} = \lim_{k\to\infty} (P^k)_{i',j} > 0.$$

Characterisation of ergodic Markov chains

Remark:
A Markov chain given by a stochastic matrix $P = (p_{i,j})_{i,j=1,\dots,n}$ is ergodic, if and only if it is **irreducible and aperiodic**.

(We skip the proof.)

Characterisation of ergodic Markov chains

Remark:
A Markov chain given by a stochastic matrix $P = (p_{i,j})_{i,j=1,\ldots,n}$ is ergodic, if and only if it is **irreducible and aperiodic**.

(We skip the proof.)

Here P is called

- **irreducible**, if the graph corresponding to P is strongly connected, and
- **aperiodic**, if every vertex i of the graph corresponding to P satisfies: the greatest common divisor of the length of all paths from i to i is 1.

Ergodic Markov chain: example

If $P = P(G, d)$ is a page rank matrix with damping factor d for some $0 \leq d < 1$, where G is a directed graph without sinks, then P is a stochastic matrix and the graph corresponding to P is the complete directed graph $\overrightarrow{K}_n$.

Ergodic Markov chain: example

If $P = P(G, d)$ is a page rank matrix with damping factor d for some $0 \leq d < 1$, where G is a directed graph without sinks, then P is a stochastic matrix and the graph corresponding to P is the complete directed graph $\overrightarrow{K}_n$.

This graph is clearly irreducible and aperiodic. Hence the page rank matrix $P(G, d)$ describes an **ergodic Markov chain**.

Properties of ergodic Markov chains: observation I

If P is a stochastic matrix that describes an ergodic Markov chain, then the following is obviously true:

1. The matrix
$$P' := \big(\lim_{k \to \infty} (P^k)_{i,j} \big)_{i,j=1,\ldots,n}$$
is well-defined (because the limits exist), and
2. All rows of P' are identical.

Properties of ergodic Markov chains: observation I

If P is a stochastic matrix that describes an ergodic Markov chain, then the following is obviously true:

1. The matrix
$$P' := \big(\lim_{k\to\infty}(P^k)_{i,j}\big)_{i,j=1,\dots,n}$$
is well-defined (because the limits exist), and
2. All rows of P' are identical.

We let $p' := (p'_1, \dots, p'_n)$ denote the first row of P'. Hence we can write P' as:

$$P' = \begin{pmatrix} p' \\ p' \\ \vdots \\ p' \end{pmatrix} = \begin{pmatrix} p'_1, \dots, p'_n \\ p'_1, \dots, p'_n \\ \vdots \\ p'_1, \dots, p'_n \end{pmatrix}.$$

Properties of ergodic Markov chains: observation I

If P is a stochastic matrix that describes an ergodic Markov chain, then the following is obviously true:

1. The matrix
$$P' := \big(\lim_{k\to\infty}(P^k)_{i,j}\big)_{i,j=1,\ldots,n}$$
is well-defined (because the limits exist), and
2. All rows of P' are identical.

We let $p' := (p'_1, \ldots, p'_n)$ denote the first row of P'. Hence we can write P' as:

$$P' = \begin{pmatrix} p' \\ p' \\ \vdots \\ p' \end{pmatrix} = \begin{pmatrix} p'_1, \ldots, p'_n \\ p'_1, \ldots, p'_n \\ \vdots \\ p'_1, \ldots, p'_n \end{pmatrix}.$$

By Part 1 we have that $P' \cdot P = P'$, and hence any distribution p' satisfies $p' \cdot P = p'$, i. e. p' is left eigenvector with eigenvalue 1 of P.

Stationary distribution: definition

A distribution Y with $Y \cdot P = Y$ is called **stationary distribution** for P.

Properties of ergodic Markov chains: observation II

Every distribution $X = (X_1, \ldots, X_n)$ satisfies: $X \cdot P' = p'$, because for every $j \in V$ the jth entry of the tuple $X \cdot P'$ is the number $\sum_{i=1}^{n} X_i \cdot p'_j = p'_j \cdot \sum_{i=1}^{n} X_i = p'_j$.

Properties of ergodic Markov chains: observation II

Every distribution $X = (X_1, \ldots, X_n)$ satisfies: $X \cdot P' = p'$, because for every $j \in V$ the jth entry of the tuple $X \cdot P'$ is the number $\sum_{i=1}^{n} X_i \cdot p'_j = p'_j \cdot \sum_{i=1}^{n} X_i = p'_j$.

Hence we have:

(a) $p' = (p'_1, \ldots, p'_n)$ is **the only stationary distribution**, and

(b) If in the graph corresponding to P the random surfer chooses the start vertex according to an arbitrary initial distribution $X = (X_1, \ldots, X_n)$ and makes sufficiently many steps, then for every $j \in V$ the probability of ending at vertex j is arbitrarily close to p'_j.

Hence the choice of the start vertex does not matter, if the random surfer surfs long enough!

Properties of ergodic Markov chains: observation III

Altogether we get:

$$p' \overset{\text{since } X\cdot P'=p'}{=} X \cdot P' \overset{\text{Part 1 of Obs. I}}{=} X \cdot \lim_{k\to\infty}(P^k) = \lim_{k\to\infty}(X \cdot P^k) = \lim_{k\to\infty} X^{(k)},$$

where $X^{(0)} := X$ and $X^{(k+1)} := X^{(k)} \cdot P$ for all $k \in \mathbb{N}$.

Properties of ergodic Markov chains: observation IV

Hence we can proceed as follows for computing an approximation of the tuple p'.

- Begin with an arbitrary distribution X
  (e. g. with the **uniform distribution** $X = (\frac{1}{n}, \ldots, \frac{1}{n})$).

Properties of ergodic Markov chains: observation IV

Hence we can proceed as follows for computing an approximation of the tuple p'.

- Begin with an arbitrary distribution X
  (e. g. with the **uniform distribution** $X = (\frac{1}{n}, \ldots, \frac{1}{n})$).

- For $k = 1, 2, 3, \ldots$ compute the tuple $X^{(k+1)} = X^{(k)} \cdot P$.

Properties of ergodic Markov chains: observation IV

Hence we can proceed as follows for computing an approximation of the tuple p'.

- Begin with an arbitrary distribution X (e. g. with the **uniform distribution** $X = (\frac{1}{n}, \ldots, \frac{1}{n})$).
- For $k = 1, 2, 3, \ldots$ compute the tuple $X^{(k+1)} = X^{(k)} \cdot P$.
- End this process, as soon as the tuple $X^{(k+1)}$ does not differ much from $X^{(k)}$, i. e. when

  $$|X_j^{(k+1)} - X_j^{(k)}| < \varepsilon$$

  holds for all $j \in \{1, \ldots, n\}$, (for a suitably chosen bound $\varepsilon > 0$).

Corollary (Solution of Problem 2)

Let $P := P(G, d)$ be the page rank matrix for a d with $0 \leq d < 1$ and a directed graph $G = (V, E)$ without sinks.

Corollary (Solution of Problem 2)

Let $P := P(G, d)$ be the page rank matrix for a d with $0 \leq d < 1$ and a directed graph $G = (V, E)$ without sinks.

We know:

- P is ergodic,
- The stationary distribution p' of P is the (uniquely determined) tuple, that has the page rank property with respect to d.

Corollary (Solution of Problem 2)

Let $P := P(G, d)$ be the page rank matrix for a d with $0 \leq d < 1$ and a directed graph $G = (V, E)$ without sinks.

We know:

- P is ergodic,
- The stationary distribution p' of P is the (uniquely determined) tuple, that has the page rank property with respect to d.

The procedure on the previous slide is an efficient method of computing an approximation of the tuple p'.

Remark I

Properties of $P(G, d)$ & the theory of Markov chains imply (for $0 \leq d < 1$):

- The sequence of the tuples $X^{(k)}$ for $k = 1, 2, 3, \ldots$ converges rapidly to p' (typically, $k \leq 1,000$ steps are sufficient.)

- There is a mathematical justification for the choice of $d = 0.85$.

Remark II

Quick computation of the vector matrix product

$$X^{(k+1)} := X^{(k)} \cdot P(G, d)$$

uses the fact that the matrix $P(G, d)$ has many identical entries of the form $\frac{1-d}{n}$.

Remark II

Quick computation of the vector matrix product

$$X^{(k+1)} := X^{(k)} \cdot P(G, d)$$

uses the fact that the matrix $P(G, d)$ has many identical entries of the form $\frac{1-d}{n}$.

Moreover, computing the vector matrix product is highly parallelisable.

Remark II

Quick computation of the vector matrix product

$$X^{(k+1)} := X^{(k)} \cdot P(G,d)$$

uses the fact that the matrix $P(G,d)$ has many identical entries of the form $\frac{1-d}{n}$.

Moreover, computing the vector matrix product is highly parallelisable.

Currently a few thousand PCs are employed, that take several hours to compute the page rank.

Remark II

Quick computation of the vector matrix product

$$X^{(k+1)} := X^{(k)} \cdot P(G, d)$$

uses the fact that the matrix $P(G, d)$ has many identical entries of the form $\frac{1-d}{n}$.

Moreover, computing the vector matrix product is highly parallelisable.

Currently a few thousand PCs are employed, that take several hours to compute the page rank.

This is surprisingly efficient, given that there are way over 1 billion webpages!

Literature

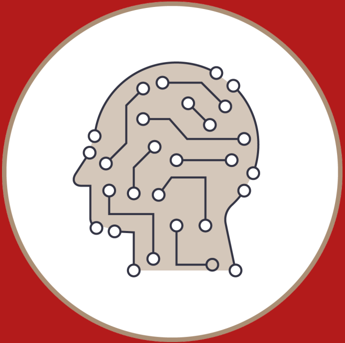
- Amy N. Langville and Carl D. Meyer. Google's Pagerank and Beyond: The Science of Search Engine Rankings. Princeton University Press, 2006.
- Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. Computer Networks, 30(1-7):107-117, 1998.
- Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. Journal of the ACM, 46(5):604-632, 1999.
- Ayman Farahat, Thomas LoFaro, Joel C. Miller, Gregory Rae, and Lesley A. Ward. Authority rankings from HITS, PageRank, and SALSA: Existence, uniqueness, and effect of initialization. SIAM Journal on Scientific Computing, 27(4):1181-1201, 2006.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Greedy algorithms: introduction

Isolde Adler and Sebastian Ordyniak



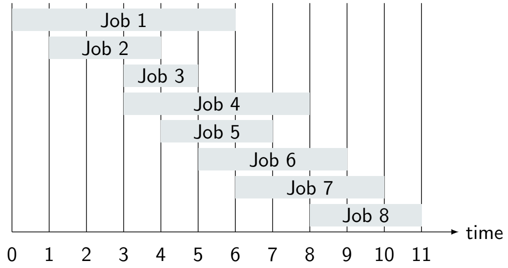
Introduction

Greedy paradigm:

- Build a solution incrementally by choosing the next solution component using a local and not forward-looking criteria.
- Greedy $\approx$ greedily take a solution component that looks locally optimal.
- Works for problems where a non-optimal solution can be improved to an optimal solution by incrementally replacing locally non-optimal solution components.
- Leads to very efficient algorithms.

Interval scheduling

- **Input:** jobs $1, \ldots, n$
- Job j starts at time s_j and finishes at time f_j.
- Two jobs are **compatible** if they do not overlap.
- **Goal:** find maximum cardinality subset of mutually compatible jobs.



Examples: Jobs 2 and 5 are compatible, jobs 2 and 3 are not compatible.

Interval scheduling: greedy template

- Consider jobs in some natural order.
- Take each job provided it is compatible with the ones already taken.

Earliest start time: Consider jobs in ascending order of s_j.

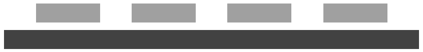
Earliest finish time: Consider jobs in ascending order of f_j.

Shortest interval: Consider jobs in ascending order of $f_j - s_j$.

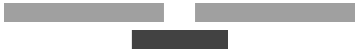
Fewest conflicts: For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

Interval scheduling: greedy template

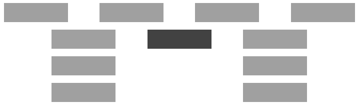
- Consider jobs in some natural order.
- Take each job provided it is compatible with the ones already taken.



counterexample for earliest start time



counterexample for shortest interval



counterexample for fewest conflicts

Earliest finish time: Counterexample? No!

Interval scheduling: greedy algorithm

- Consider jobs in increasing order of finish time.
- Take each job provided it is compatible with the ones already taken.

Implementation: A is the set of selected jobs.

```
Sort jobs by finish times such that $f_1 \leq f_2 \leq \cdots \leq f_n$;
$A \leftarrow \varnothing$;
for $j \leftarrow 1$ to $n$ do
    if job $j$ compatible with all jobs in $A$ then
        $A \leftarrow A \bigcup \{j\}$
return $A$;
```

Interval scheduling: greedy algorithm

Implementation: runtime in $\mathcal{O}(n \log n)$.

- Jobs are sorted by finish time and renumbered accordingly. If $f_i \leq f_j$, then $i < j$. The sorting runs in $\mathcal{O}(n \log n)$ time.
- Jobs are chosen in order of ascending f_i.
- Let t be the finish time of the current job:
  - Then take the first job j in order such that: $s_j \geq t$.
  - This job becomes the new current job and the search is continued with this job.
- The algorithm requires only one run through the jobs $\mathcal{O}(n)$.
- Together with the sort the required time is therefore $\mathcal{O}(n \log n)$.

Interval scheduling: greedy algorithm

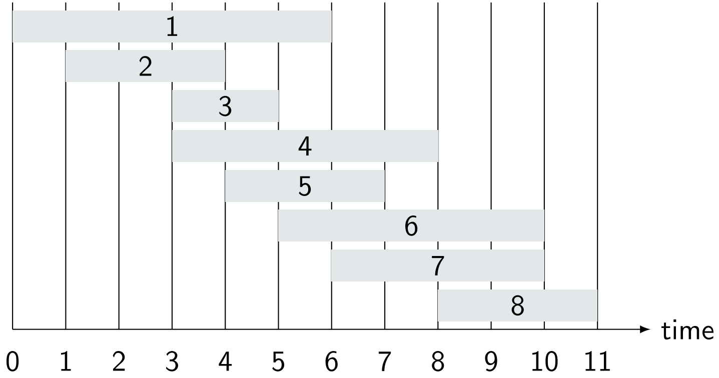
Detailed pseudocode with efficient test for compatiblility with the jobs in A:

```
sort jobs by finish time such that $f_1 \leq f_2 \leq \cdots \leq f_n$;
$A \leftarrow \varnothing$;
$t \leftarrow 0$;
for $j \leftarrow 1$ to $n$ do
    if $t \leq s_j$ then
        $A \leftarrow A \cup \{j\}$;
        $t \leftarrow f_j$;
return $A$;
```

Interval scheduling: example

Job	**2**	**3**	**1**	**5**	**4**	**6**	**7**	**8**
s_i	1	3	0	4	3	5	6	8
f_i	4	5	6	7	8	10	10	11

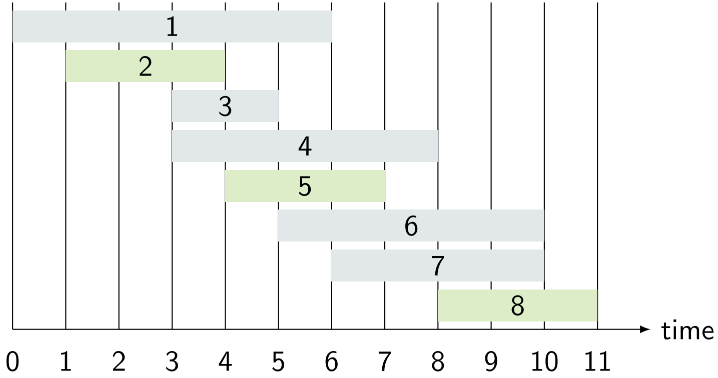
Jobs sorted by finish time.



Interval scheduling: example

Job	**2**	**3**	**1**	**5**	**4**	**6**	**7**	**8**
s_i	1	3	0	4	3	5	6	8
f_i	4	5	6	7	8	10	10	11

Jobs sorted by finish time.



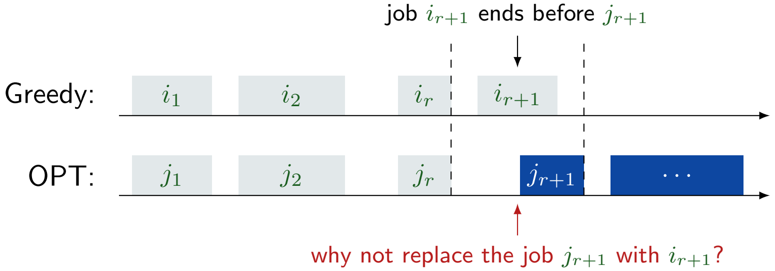
Solution: Jobs 2, 5 and 8

Interval scheduling: analysis

Theorem: The greedy algorithm is correct, i. e., always outputs an optimal solution.

Proof: (by contradiction)

- Assume the algorithm does not return an optimal solution.
- Let $i_1, i_2, \ldots, i_k$ be the set of jobs chosen by the algorithm.
- Let $j_1, j_2, \ldots, j_m$ be the set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for largest possible r.

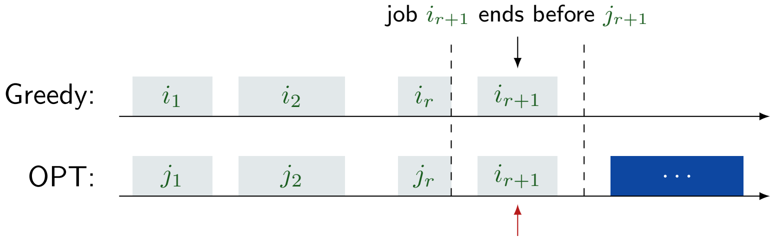

job i_{r+1} ends before j_{r+1}
Greedy:
i_1
i_2
i_r
i_{r+1}
OPT:
j_1
j_2
j_r
j_{r+1}
$\cdots$
why not replace the job j_{r+1} with i_{r+1}?

Interval scheduling: analysis

Theorem: The greedy algorithm is correct, i. e., always outputs an optimal solution.

Proof: (by contradiction)

- Assume the algorithm does not return an optimal solution.
- Let $i_1, i_2, \ldots, i_k$ be the set of jobs chosen by the algorithm.
- Let $j_1, j_2, \ldots, j_m$ be the set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for largest possible r.



Solution is still possible and optimal, however, contradicts the maximality of r.

Greedy algorithm: general idea

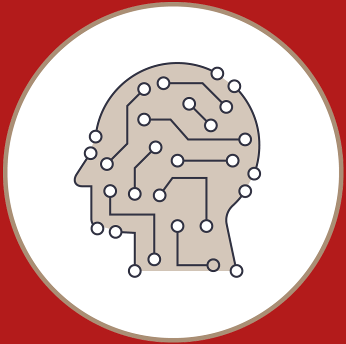
- A solution is built iteratively; at every step the problem is reduced to a smaller problem.
- **Greedy principal:** Always add a locally most attractive solution component.
- Once a decision is made it is not taken back.
- Usually easy to construct and implement.
- Can result in an optimal solution, but not always does.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

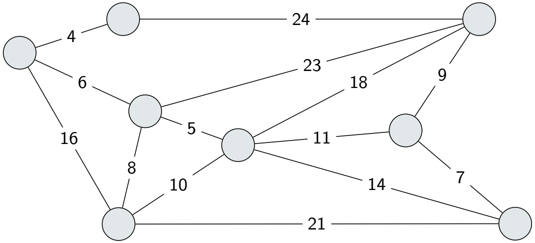
Greedy algorithms: minimum spanning tree problem

Isolde Adler and Sebastian Ordyniak

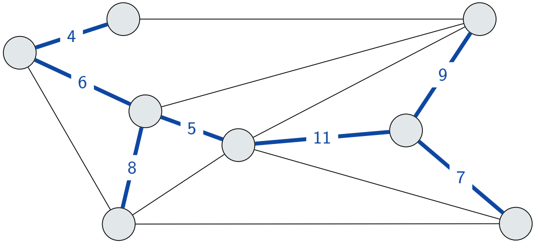


Minimum spanning tree (MST)

Minimum spanning tree: Given a connected (undirected) graph $G = (V, E)$ with positive rational edge weights $w\colon E \to \mathbb{Q}_{\geq 0}$, an MST $T = (V, F)$ is a spanning tree, whose sum of edge weights is minimized.



$G = (V, E)$



$\mathbf{T}, \sum_{e \in F} w(e) = 50$

🟩 *spanning tree ≈ a subgraph of a graph that is a tree and spans (contains) all vertices of the graph.*

Minimum spanning tree problem

MST problem

Input: An undirected graph $G = (V, E)$ with weights $w \colon E \to \mathbb{Q}_{\geq 0}$.

Question: A spanning tree $T = (V, F)$ that minimizes $\sum\limits_{e \in F} w(e)$.

Remarks:

- A graph can have exponentially many spanning trees, e.g., the complete graph K_n on n vertices has n^{n-2} spanning trees (Cayley's theorem).
- Therefore, it is not possible to solve the MST problem by enumerating all possible spanning trees.
- Fortunately, there exist much more efficient algorithms for the problem that employ the Greedy approach.

Applications

MST is a fundamental problem with diverse applications.

- Network design:
  - Telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems:
  - Traveling salesperson problem, Steiner tree
- Indirect applications:
  - Max bottleneck paths
  - LDPC codes for error correction
  - Image registration with Renyi entropy
  - Learning salient features for real-time face verification
  - Reducing data storage in sequencing amino acids in a protein
  - Model locality of particle interactions in turbulent fluid flows
  - Autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

Greedy algorithms

Kruskal's algorithm. Start with $E(T) = \varnothing$. Consider edges in ascending order of weight. Insert edge e in T unless doing so would create a cycle.

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add a cheapest edge e to T that has exactly one endpoint in T.

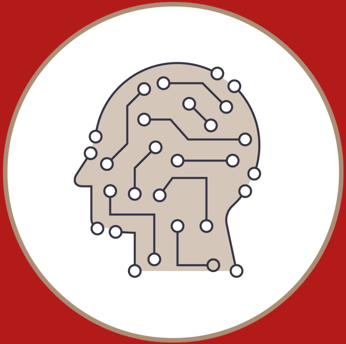
Remark. Both algorithms produce an MST.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Greedy algorithms: Prim's algorithm

Isolde Adler and Sebastian Ordyniak

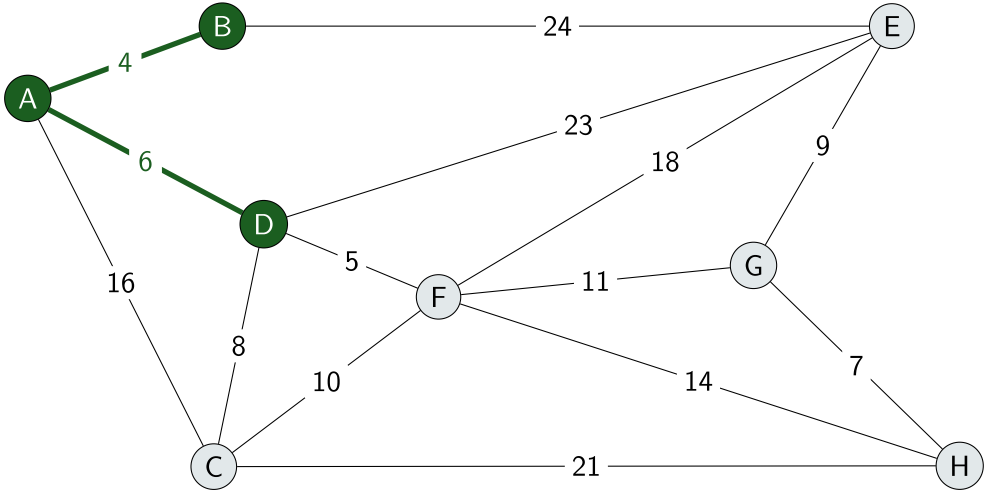


Prim's algorithm: main idea

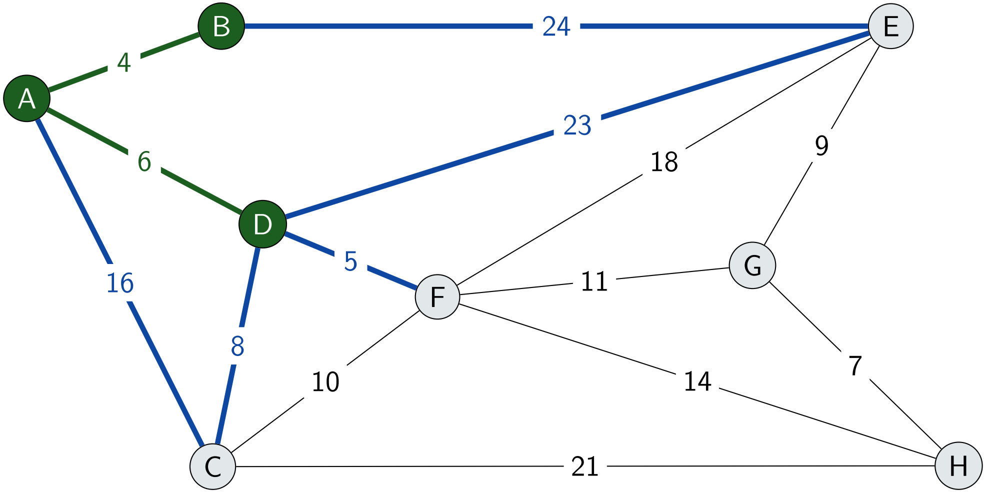
Main Idea:

- Start with node s and greedily grow a tree T from s outward.
- At each step, add a cheapest edge e to T that has exactly one endpoint in T.

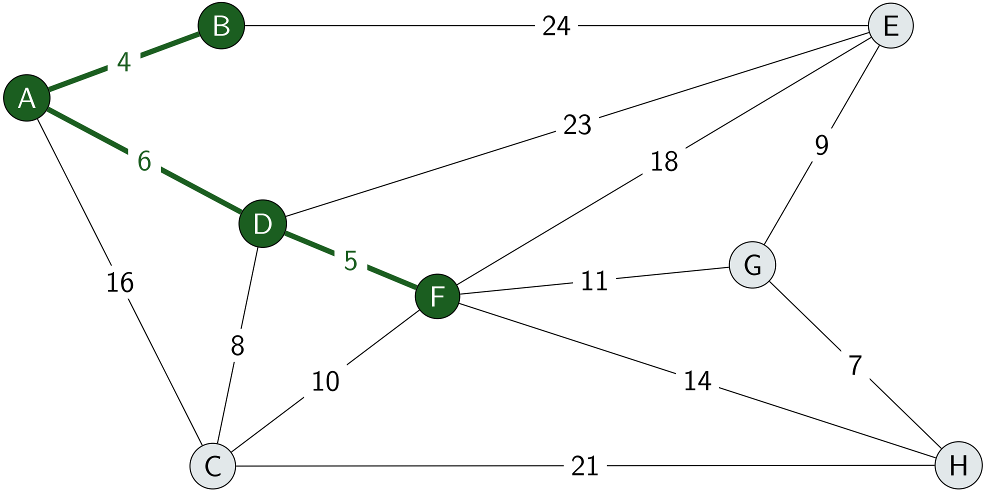
Prim's algorithm: growing the tree



Prim’s algorithm: growing the tree



Prim's algorithm: growing the tree



Prim's algorithm: outline

(I0) Iteratively grow a set S of **visited vertices**, for which we have already calculated an MST T.

(S1) Initialize $S = \{s\}$.

(S2) Choose a cheapest edge e with exactly one endpoint in S.

(S3) Add e to T and add the endpoint of e that is not in S to S.

(S4) Continue with step (S2).

Prim's algorithm: improvement

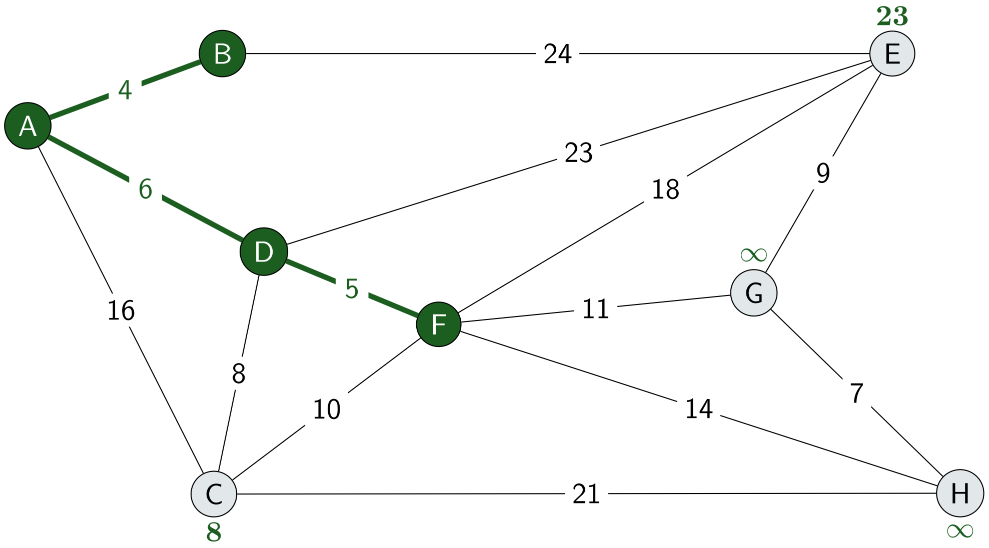
To avoid having to go over all edges with exactly one endpoint in S at every step of the algorithm, we can store the currently cheapest connection from S to every vertex outside of S and only update when a new vertex is added to S, i. e.:

- We store the cheapest connection using an array $a[u]$ for every vertex u outside of S.
- We initially set $S = \varnothing$ and set $a[u]$ to ∞ for every $u \in V$.
- Every time a vertex v is added to S we set:

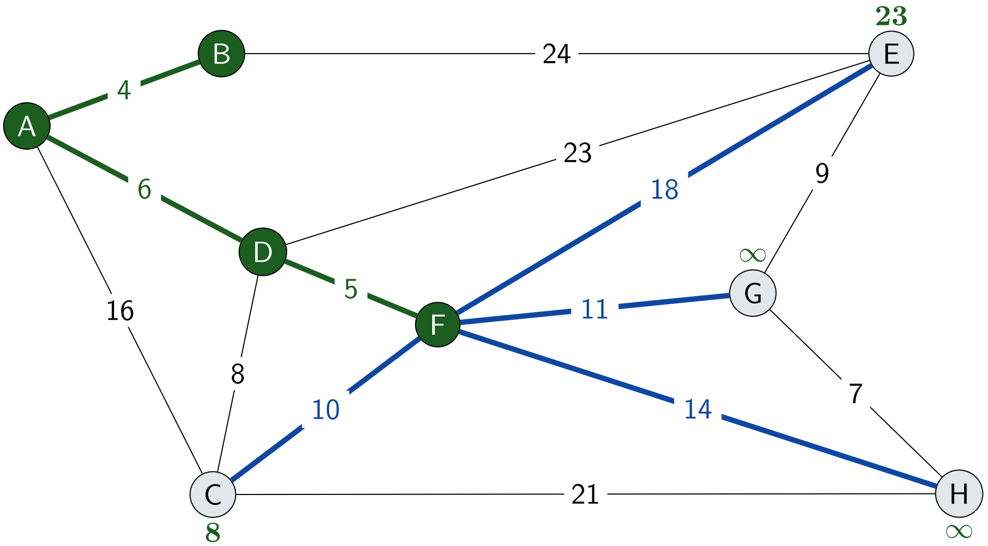
  $$a[u] = \min\{a[u], w(v, u)\}$$

  for every vertex $u \in N(v) \setminus S$.
- The next vertex to be added to S is then always a vertex that minimizes $a[u]$ among all vertices outside of S.

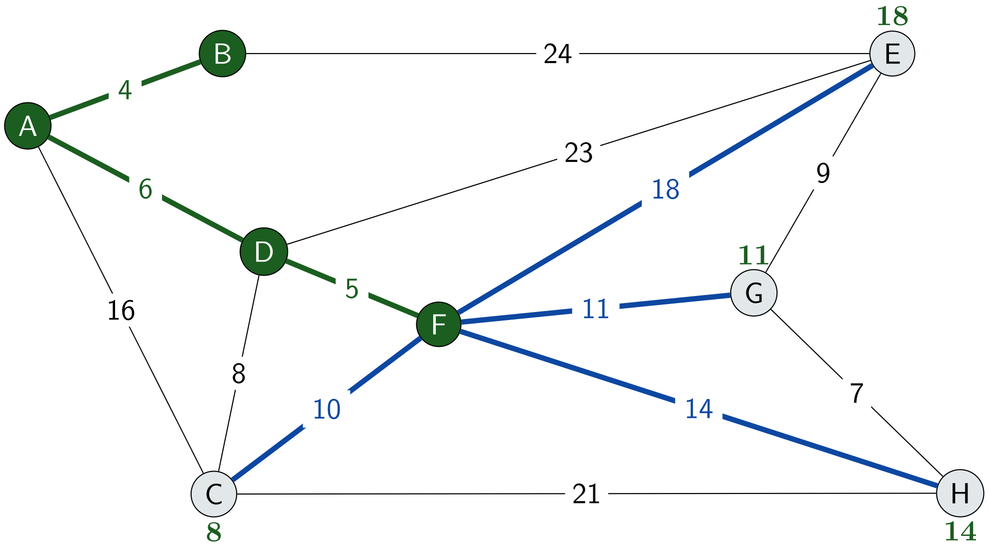
Prim's algorithm: update improvement



Prim's algorithm: update improvement



Prim's algorithm: update improvement

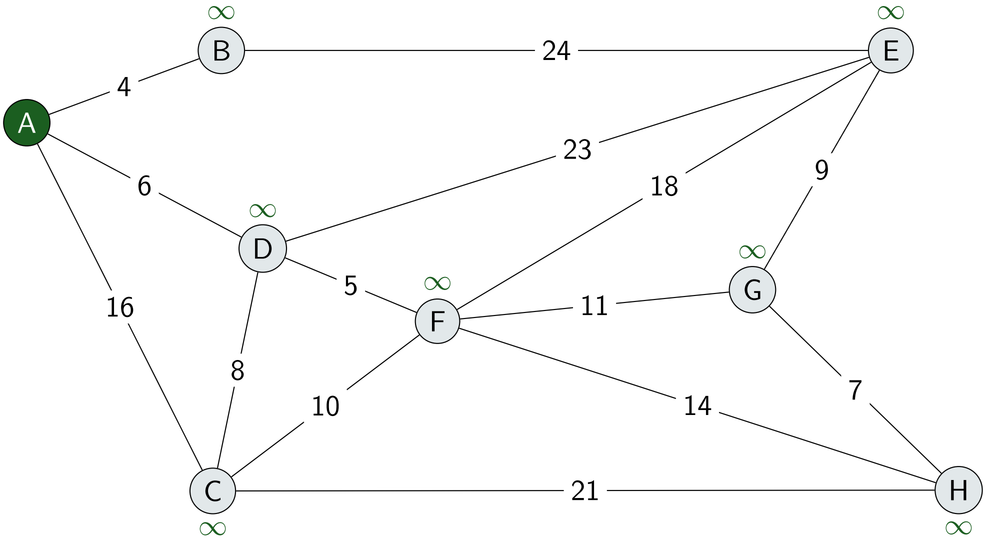


Implementation: Prim's Algorithm

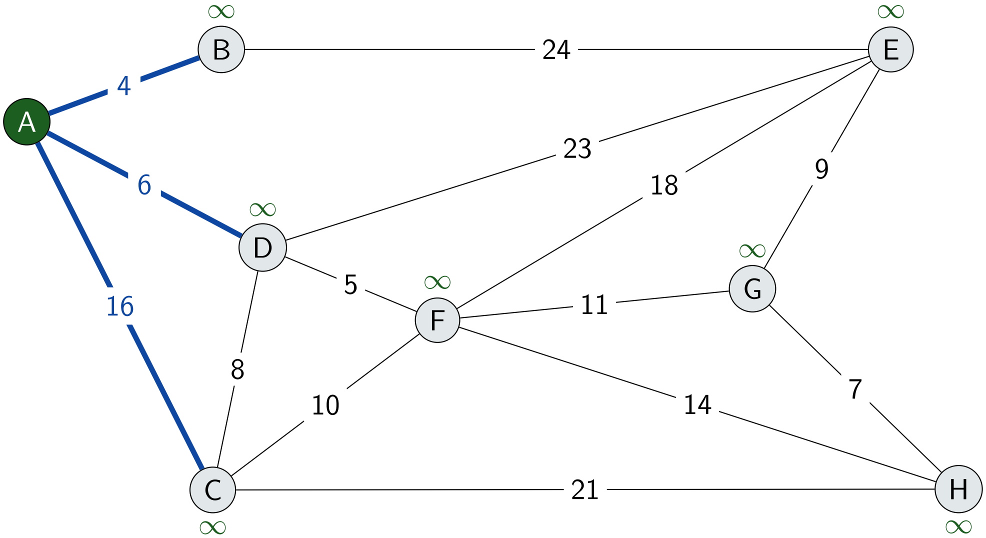
```
input  : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$
output : minimum spanning tree $(V, F)$ of $G$

procedure Prim$(G, w)$
    choose an arbitrary start node $s \in V$;$L \leftarrow V \setminus \{s\}$ ;        // $L$ is a list/set
    for $v \in L$ do visited$[v]$ ←false;$a[v] \leftarrow \infty$;
    ;
    for $n \in N(s)$ do $a[n] \leftarrow w(s, n)$; $p[n] \leftarrow s$;
    ;
    visited$[s]$ ← true; $F \leftarrow \varnothing$;
    while $L$ is not empty do
        $u \leftarrow$ delete a minimum (w.r.t. $a[u]$) element from $L$;
        visited$[u]$ ←true; $F \leftarrow F \cup \{\{p[u], u\}\}$;
        for $v \in N(u)$ do
            if !visited$[v]$ and $a[v] > w(u, v)$ then
                $p[v] \leftarrow u$; $a[v] \leftarrow w(u, v)$;
```

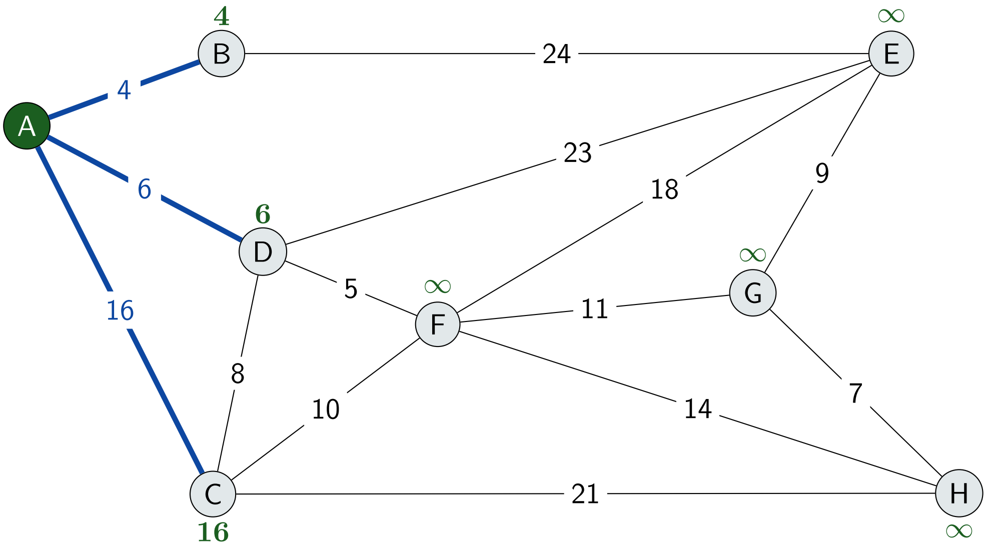
Prim's algorithm: example



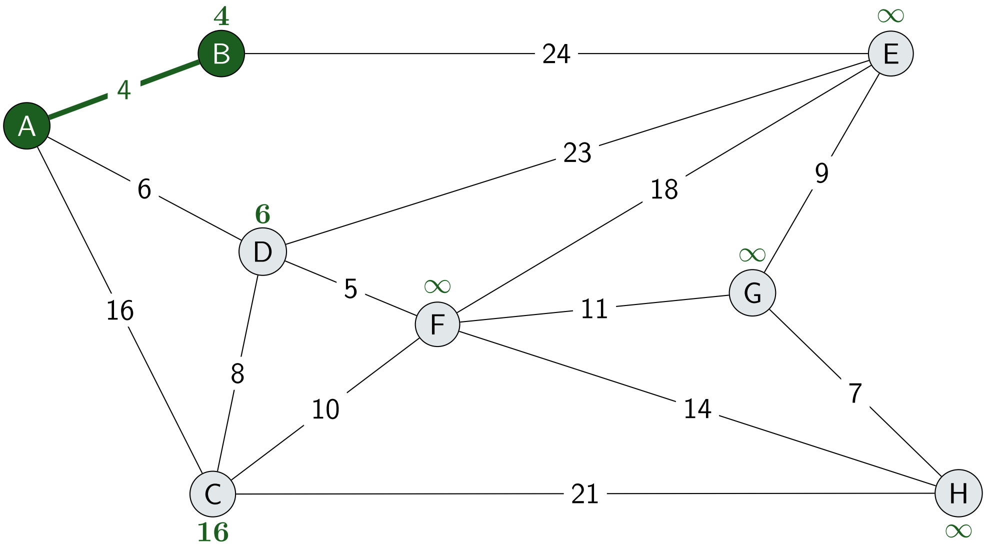
Prim's algorithm: example



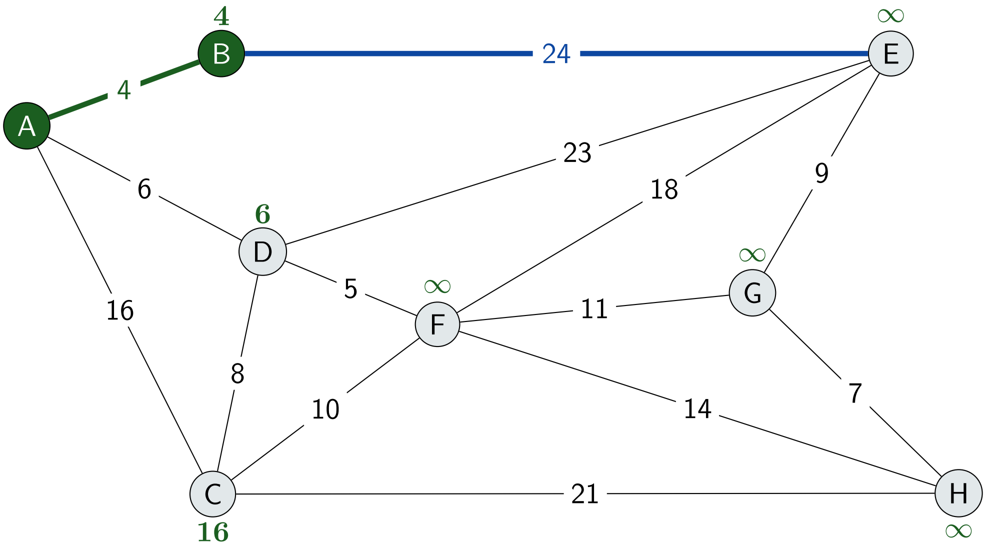
Prim's algorithm: example



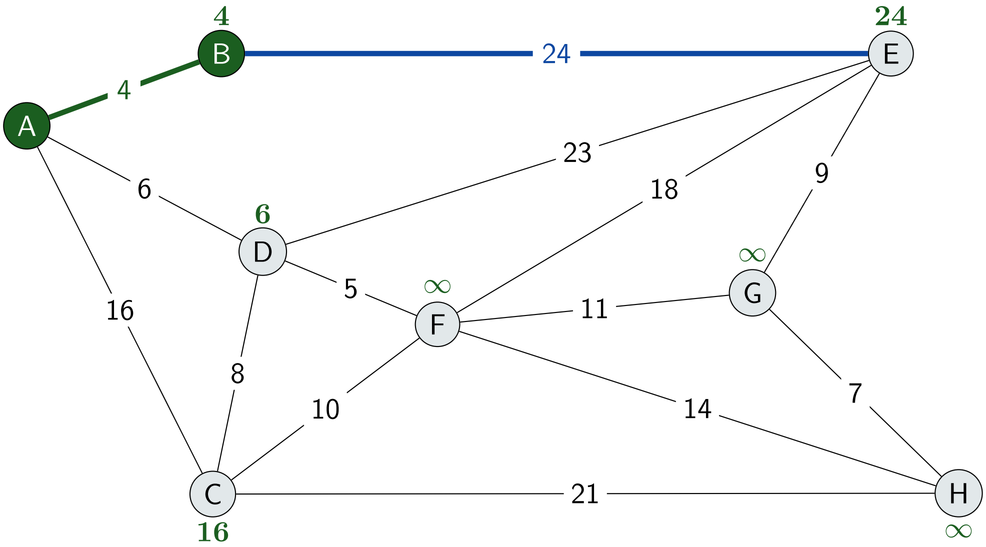
Prim's algorithm: example



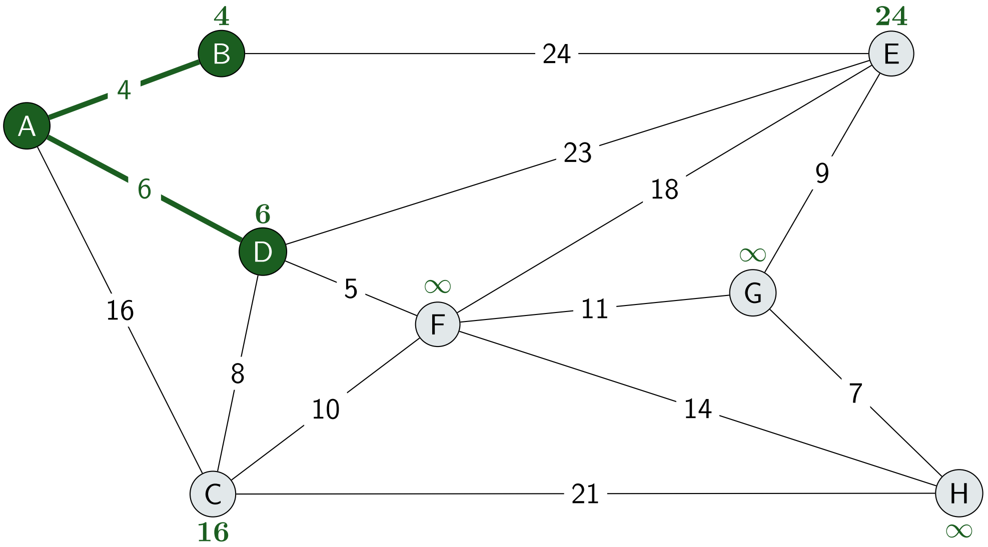
Prim's algorithm: example



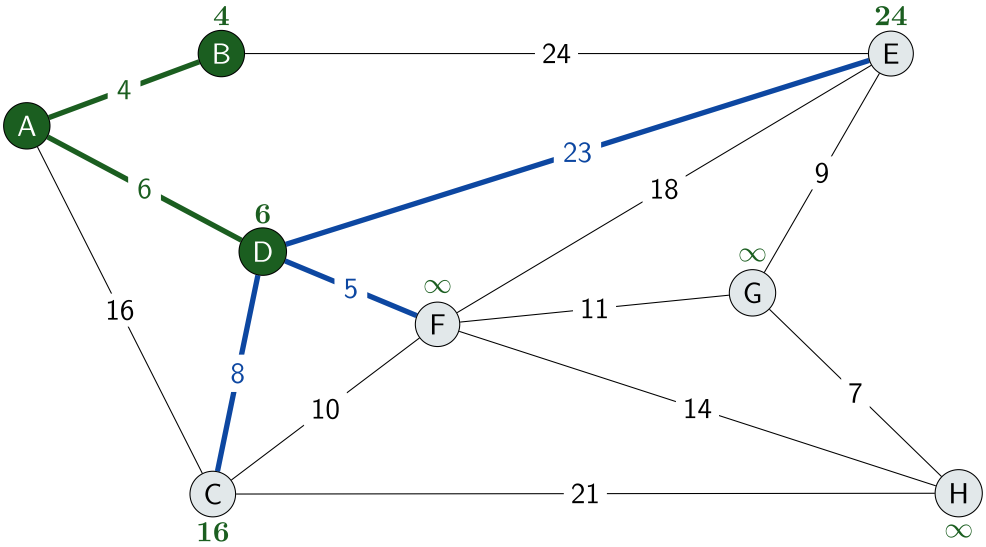
Prim's algorithm: example



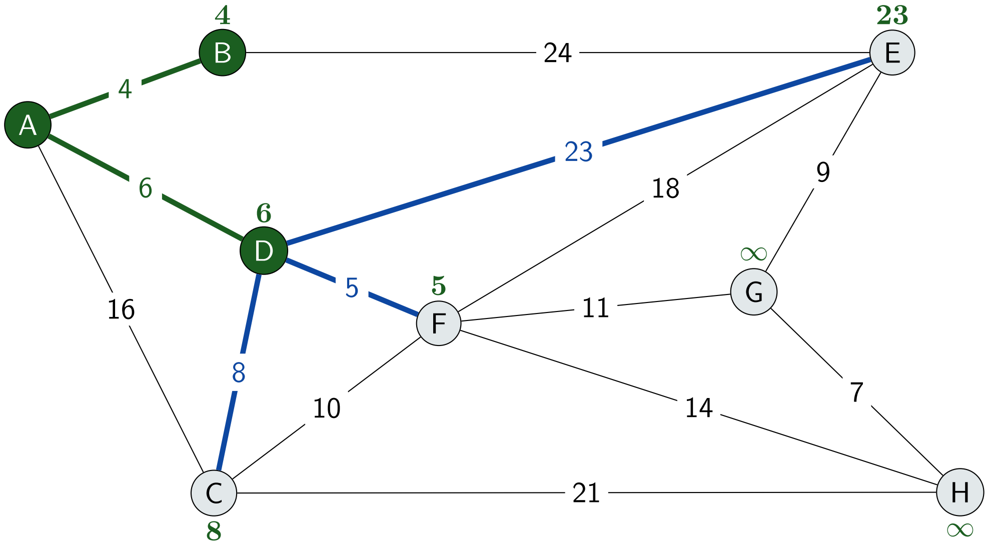
Prim's algorithm: example



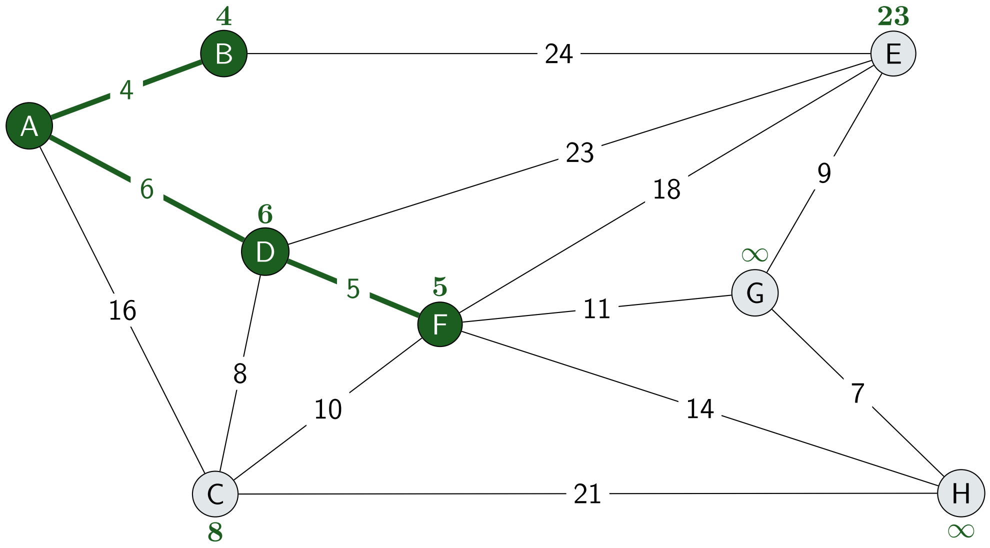
Prim's algorithm: example



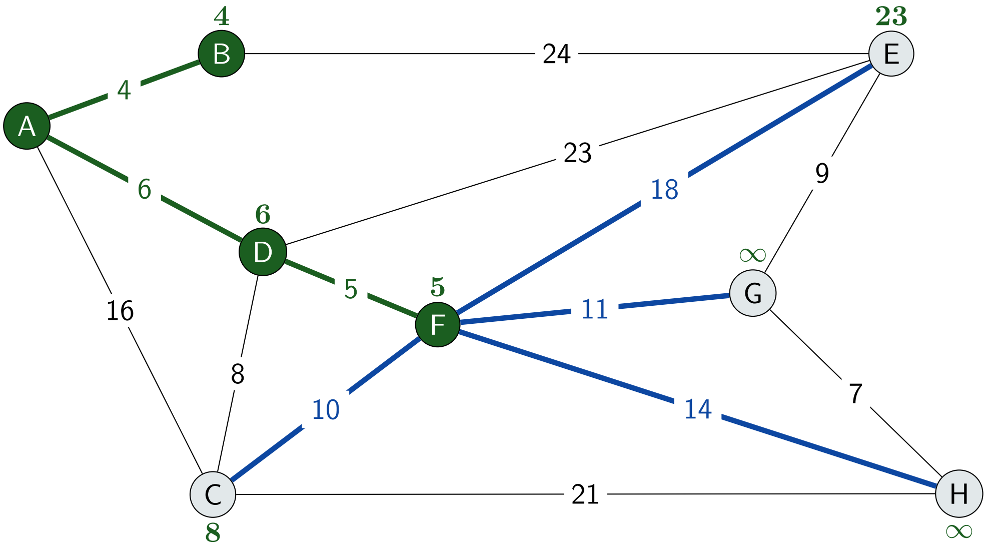
Prim's algorithm: example



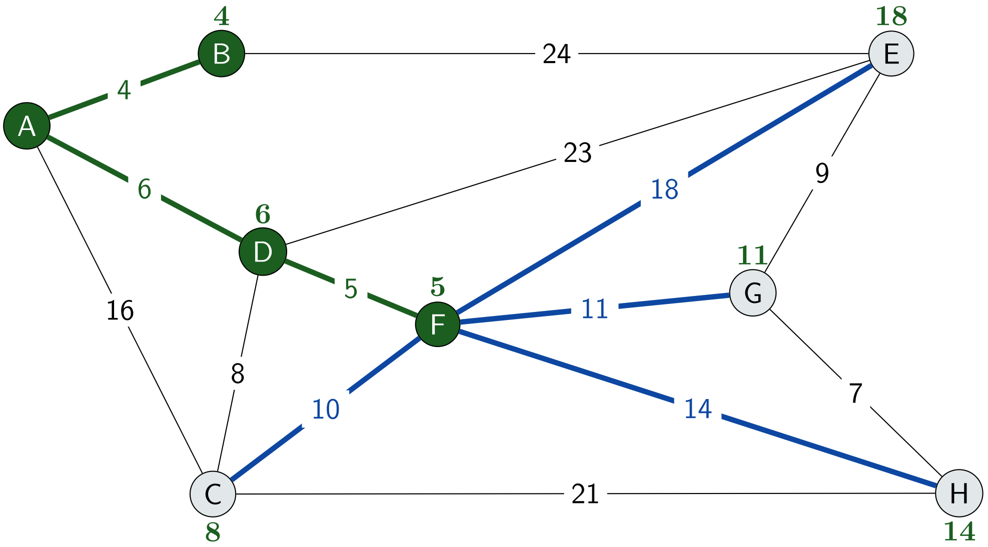
Prim's algorithm: example



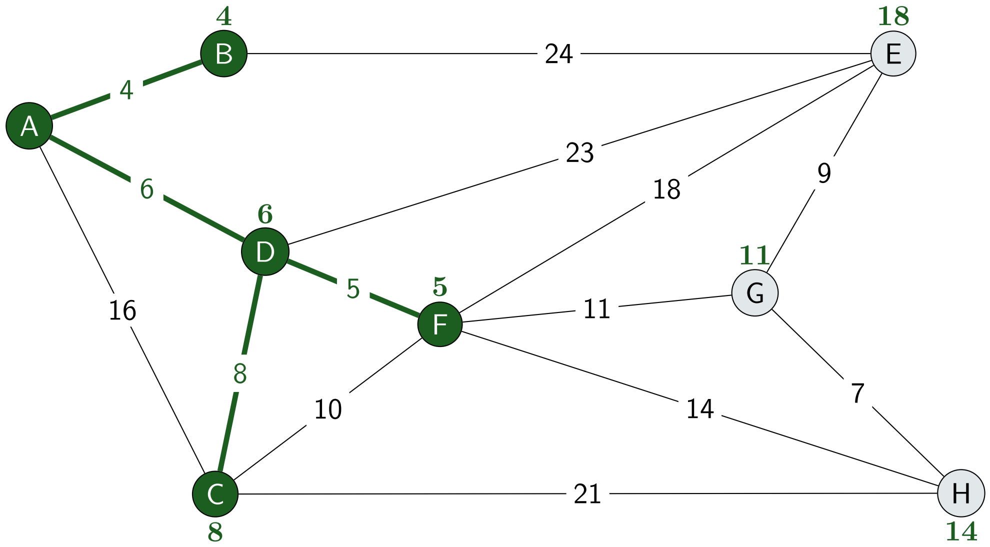
Prim's algorithm: example



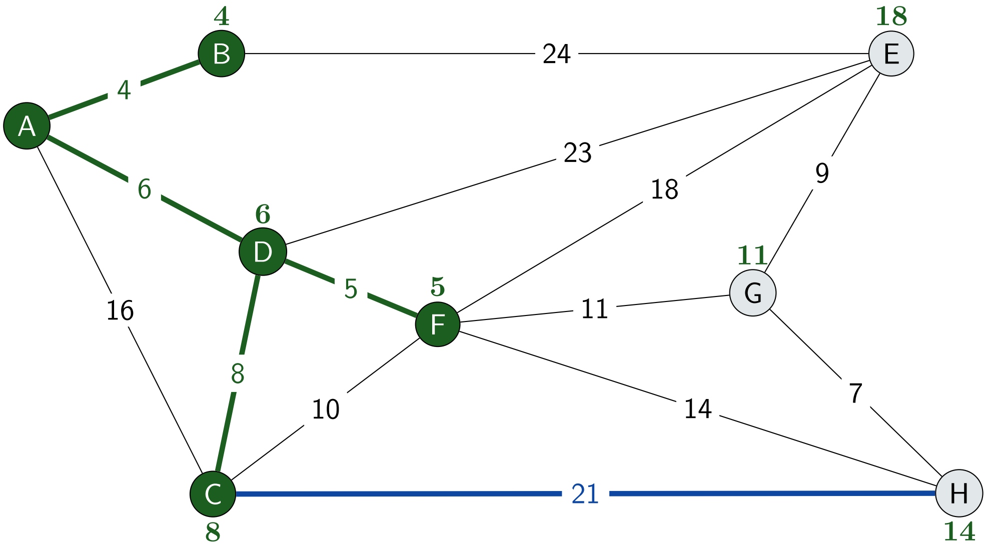
Prim's algorithm: example



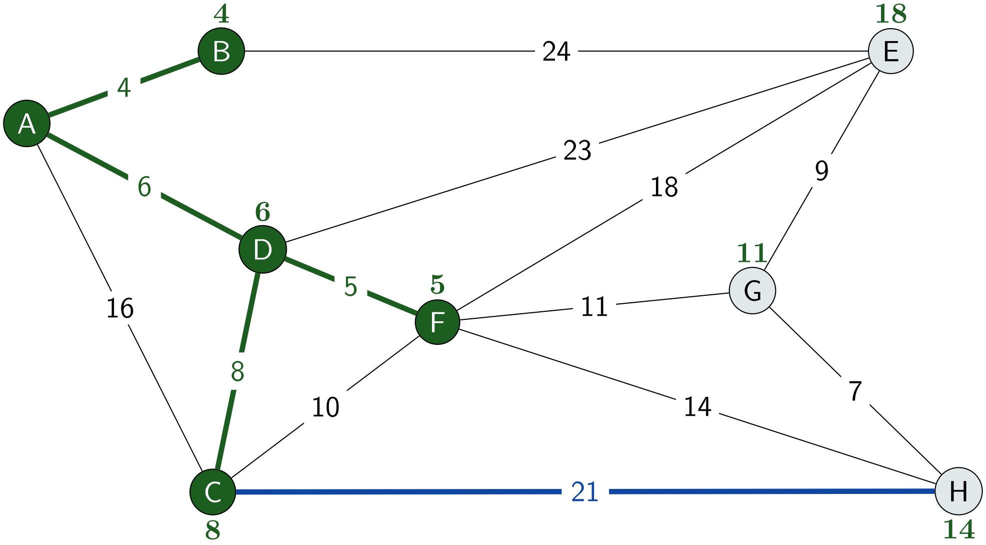
Prim's algorithm: example



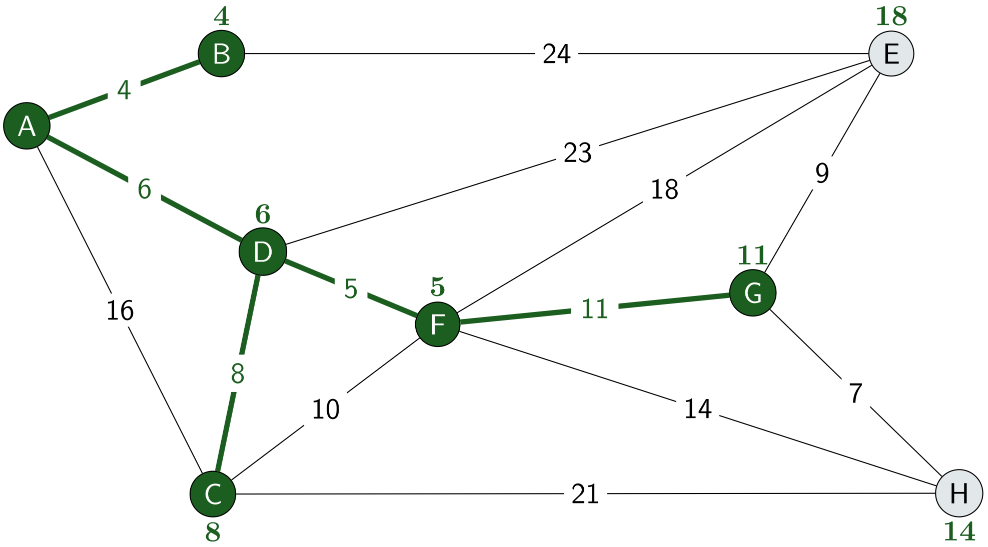
Prim's algorithm: example



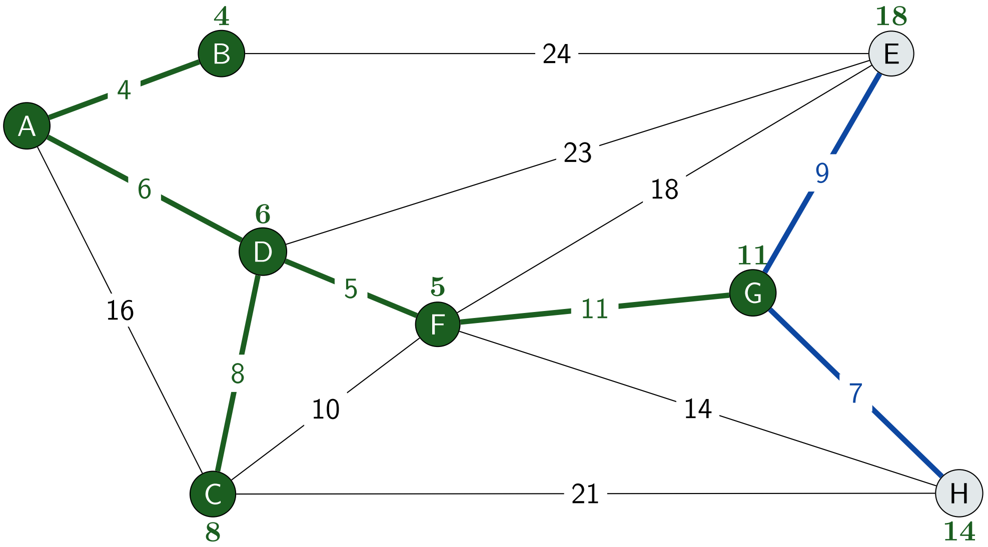
Prim's algorithm: example



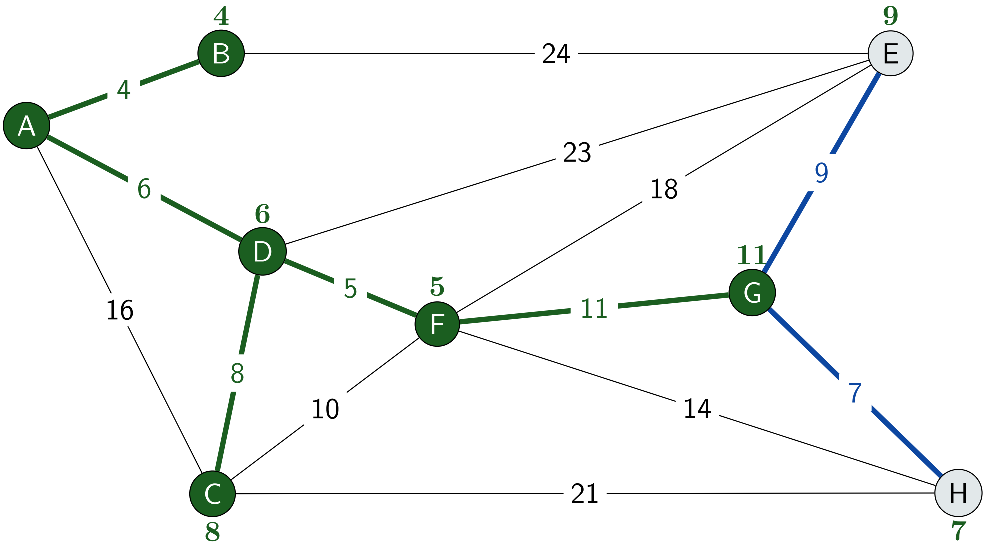
Prim's algorithm: example



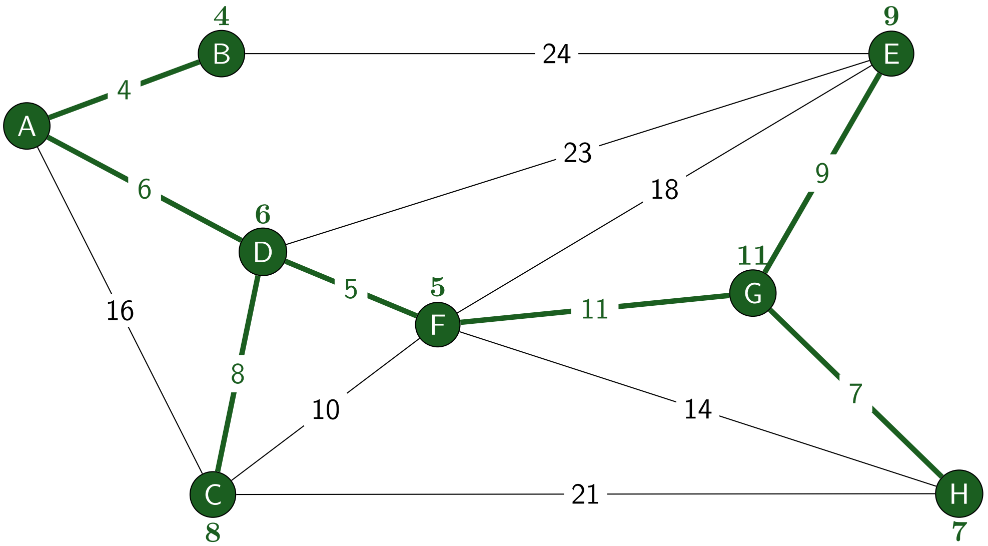
Prim's algorithm: example



Prim's algorithm: example



Prim’s algorithm: example



Prim's algorithm: running time

```
input  : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$
output : minimum spanning tree $(V, F)$ of $G$

procedure Prim$(G, w)$
    choose an arbitrary start node $s \in V$;$L \leftarrow V \setminus \{s\}$ ;        // $\mathcal{O}(n)$
    for $v \in L$ do visited$[v]$ ←false;$a[v] \leftarrow \infty$;
    ;                                                                                     // $\mathcal{O}(n)$
    for $n \in N(s)$ do $a[n] \leftarrow w(s, n)$; $p[n] \leftarrow s$;
    ;                                                                                     // $\mathcal{O}(n)$
    visited$[s]$ ← true; $F \leftarrow \varnothing$ ;                                      // $\mathcal{O}(1)$
    while $L$ is not empty do                                                             // $\mathcal{O}(n)\times$
        $u \leftarrow$ delete a minimum (w.r.t. $a[u]$) element from $L$ ;                  // $\mathcal{O}(n)$
        visited$[u]$ ←true; $F \leftarrow F \cup \{\{p[u], u\}\}$ ;                        // $\mathcal{O}(1)$
        for $v \in N(u)$ do                                                  // $\mathcal{O}(m)\times$ overall
            if !visited$[v]$ and $a[v] > w(u, v)$ then                                     // $\mathcal{O}(1)$
                $p[v] \leftarrow u$; $a[v] \leftarrow w(u, v)$ ;                           // $\mathcal{O}(1)$
```

Prim's algorithm: running time

```
input  : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$
output : minimum spanning tree $(V, F)$ of $G$

procedure Prim$(G, w)$
    choose an arbitrary start node $s \in V$;$L \leftarrow V \setminus \{s\}$ ;                // $\mathcal{O}(n)$
    for $v \in L$ do visited$[v]$ ←false;$a[v] \leftarrow \infty$;
    ;                                                                                       // $\mathcal{O}(n)$
    for $n \in N(s)$ do $a[n] \leftarrow w(s, n)$; $p[n] \leftarrow s$;
    ;                                                                                       // $\mathcal{O}(n)$
    visited$[s]$ ← true; $F \leftarrow \varnothing$ ;                                       // $\mathcal{O}(1)$
    while $L$ is not empty do                                                               // $\mathcal{O}(n)\times$
        $u \leftarrow$ delete a minimum (w.r.t. $a[u]$) element from $L$ ;                   // $\mathcal{O}(n)$
        visited$[u]$ ←true; $F \leftarrow F \cup \{\{p[u], u\}\}$ ;                         // $\mathcal{O}(1)$
        for $v \in N(u)$ do                                                   // $\mathcal{O}(m)\times$ overall
            if !visited$[v]$ and $a[v] > w(u, v)$ then                                      // $\mathcal{O}(1)$
                $p[v] \leftarrow u$; $a[v] \leftarrow w(u, v)$ ;                            // $\mathcal{O}(1)$
```

Intermezzo: priority queues

Priority Queue:

- A data-structure that maintains a set U of elements.
- Each element $u \in U$ has an associated value, which describes its priority (in our case $a[u]$).
- Smaller values represent higher priorities.

Supported Operations:

- `initializePriorityQueue`(U, a):
  Initializes and returns a priority queue with the elements in the set U using the priorities given by the array a, i. e., $a[u]$ is the priority of $u \in U$.

- `extractMin(`Q`)`:
  Removes and returns an element in Q with minimum priority.

- `decreaseKey(`Q,u,p`)`:
  Sets the priority of the element u in Q to p.

Prim's Algorithm: implementation using priority queue Q

```
input  : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$
output : minimum spanning tree $(V, F)$ of $G$

procedure Prim$(G, w)$
    choose an arbitrary start node $s \in V$;visited$[s] \leftarrow$ true; $F \leftarrow \varnothing$;
    for $v \in L$ do visited$[v] \leftarrow$false;$a[v] \leftarrow \infty$;
    ;
    for $n \in N(s)$ do $a[n] \leftarrow w(s,n)$; $p[n] \leftarrow s$;
    ;
    $Q \leftarrow$initializePriorityQueue$(V \setminus \{s\}, a)$;
    while $L$ is not empty do
        $u \leftarrow$ extractMin(Q);
        visited$[u] \leftarrow$true; $F \leftarrow F \cup \{\{p[u], u\}\}$;
        for $v \in N(u)$ do
            if !visited$[v]$ and $a[v] > w(u, v)$ then
                $p[v] \leftarrow u$;decreaseKey(Q,v,w(u,v));
```

Prim's Algorithm: implementation using priority queue Q

```
input   : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$
output  : minimum spanning tree $(V, F)$ of $G$

procedure Prim$(G, w)$
    choose an arbitrary start node $s \in V$;visited$[s] \leftarrow$ true; $F \leftarrow \varnothing$;
    for $v \in L$ do visited$[v] \leftarrow$false;$a[v] \leftarrow \infty$;
    ;
    for $n \in N(s)$ do $a[n] \leftarrow w(s, n)$; $p[n] \leftarrow s$;
    ;
    $Q \leftarrow$initializePriorityQueue$(V \setminus \{s\}, a)$;                // $1\times$
    while $L$ is not empty do
        $u \leftarrow$ extractMin(Q);                                          // $n\times$
        visited$[u] \leftarrow$true; $F \leftarrow F \cup \{\{p[u], u\}\}$;
        for $v \in N(u)$ do
            if !visited$[v]$ and $a[v] > w(u, v)$ then
                $p[v] \leftarrow u$;decreaseKey(Q,v,w(u,v));                  // $m\times$
```

Prim's algorithm: complexity

Let $n = |V|$ and $m = |E|$. The total running time of the algorithm depends on the implementation of the priority queue Q.

Operation		Queue Implementation		
Name	$\#$	**List**	**Minimum Heap**	**Fibonacci Heap**[1]
`decreaseKey`	m	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
`extractMin`	n	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`initializePriorityQueue`	1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Total		$\mathcal{O}(n^2 + m)$ $= \mathcal{O}(n^2)$	$\mathcal{O}((n+m)\log n)$	$\mathcal{O}(n \log n + m)$

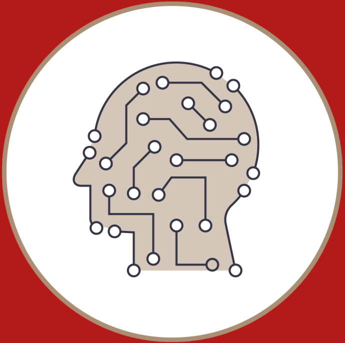
[1] amortized complexity

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Greedy algorithms: Kruskal’s algorithm

Isolde Adler and Sebastian Ordyniak



Kruskal's algorithm: main idea

Main idea:

- Build the tree T iteratively by adding one edge at every step starting from the empty tree.
- Consider the edges in ascending order with respect to weight.
- At each step check whether the current edge e creates a cycle when adding it to the already build forest T.
  - If not, then add the edge e to T.
  - Continue with the next edge in the ordering.

Observations: Adding edge $e = \{u, v\}$ to a forest T creates a cycle if and only if u and v are in the same component of T.

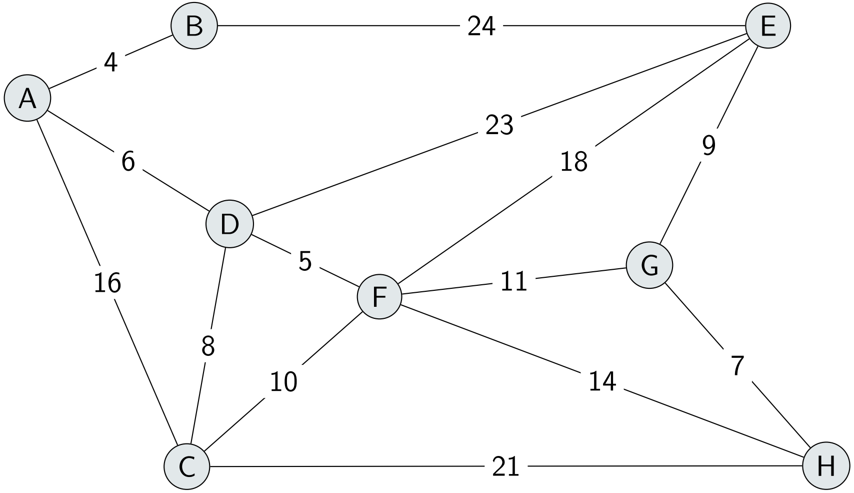
Kruskal's algorithm: implementation

```
input  : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$.
output : minimum spanning tree $(V, F)$ of $G$

procedure Kruskal $(G, w)$
    sort edges by ascending by weight such that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$;
    $F \leftarrow \varnothing$;
    for $i = 1$ to $m$ do
        $\{u, v\} \leftarrow e_i$;
        if $u$ and $v$ are in different components of $(V, F)$ then  // check with BFS
            $F \leftarrow F \cup \{e_i\}$;
```

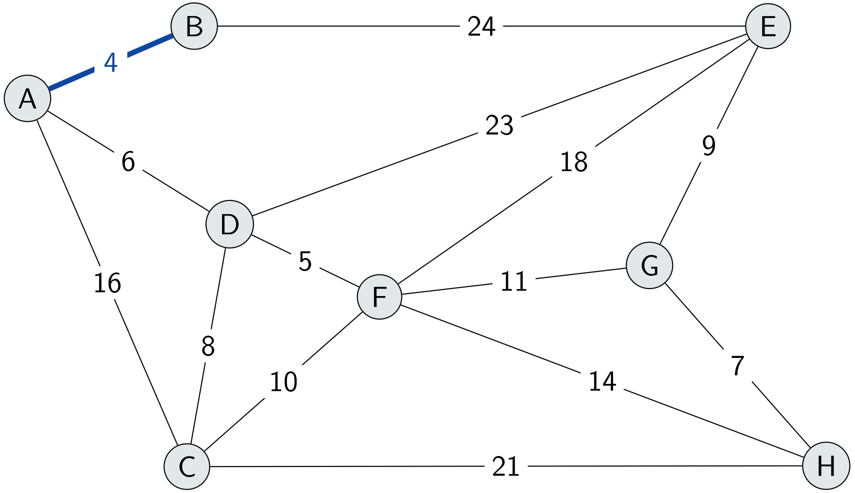
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



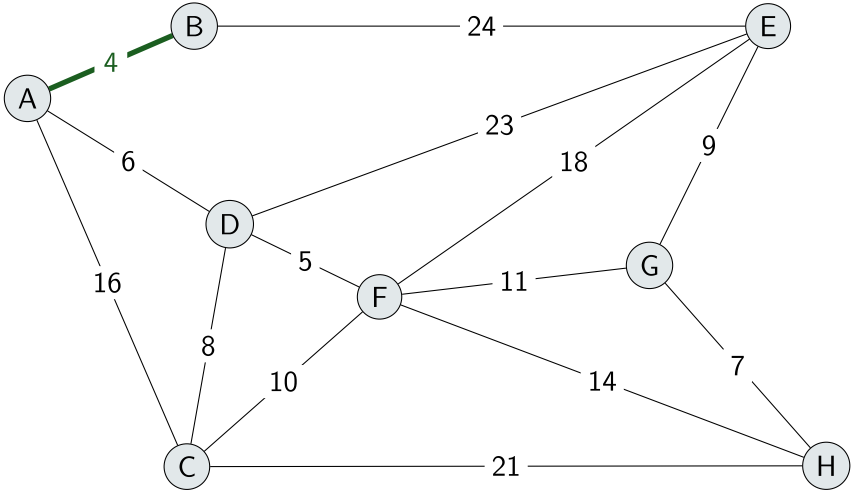
Kruskal's algorithm: example

1. **A-B (4)**
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



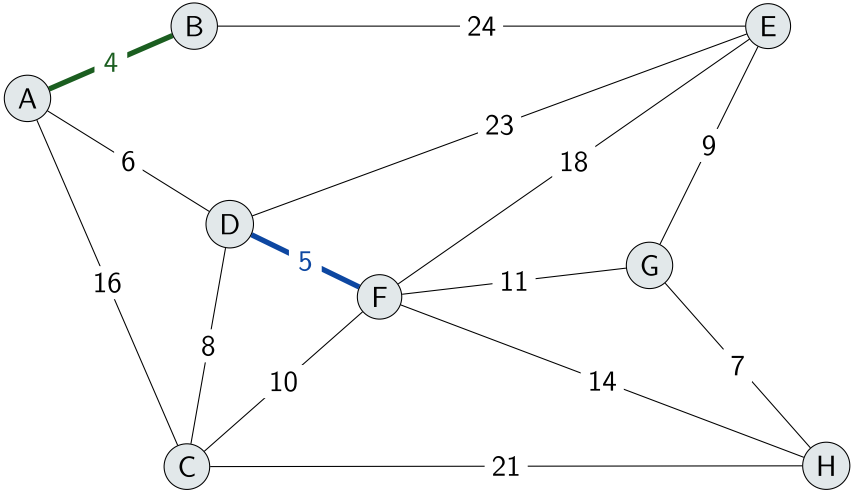
Kruskal's algorithm: example

1. **A-B (4)**
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. ...



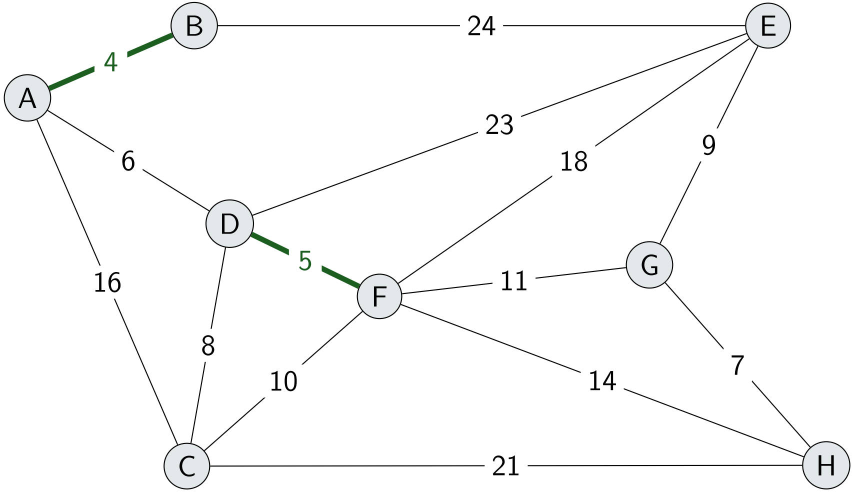
Kruskal's algorithm: example

1. A-B (4)
2. **D-F (5)**
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



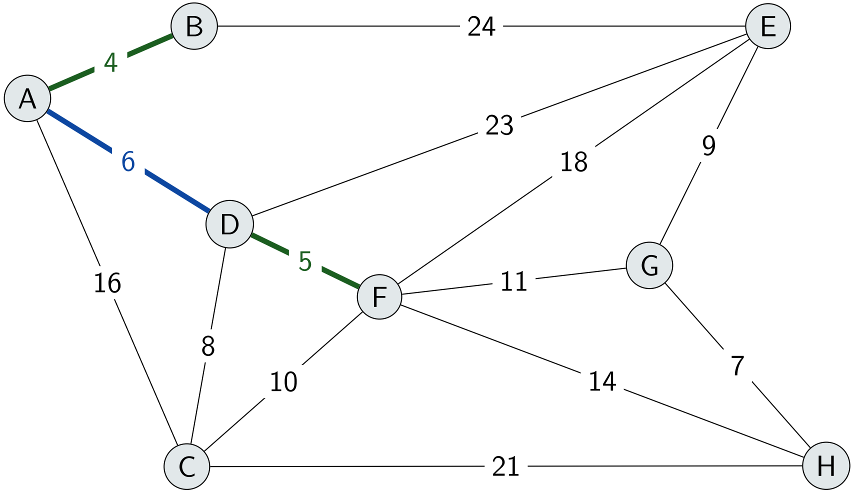
Kruskal's algorithm: example

1. A-B (4)
2. **D-F (5)**
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



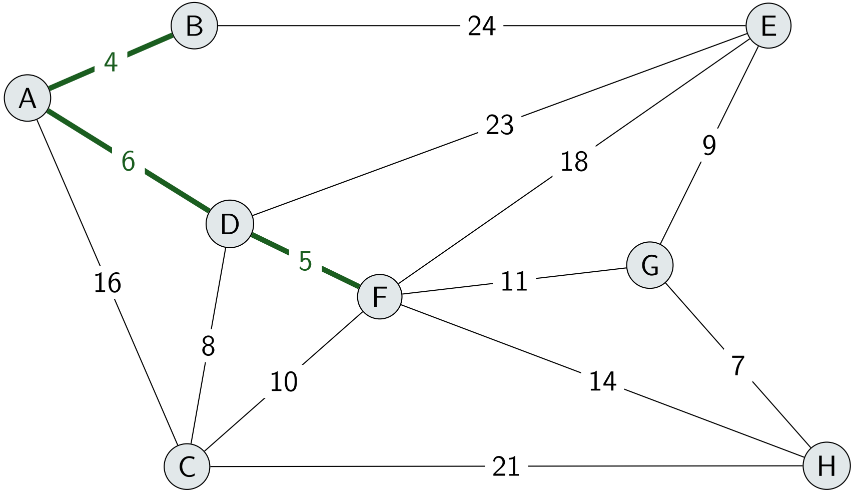
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. **A-D (6)**
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. ...



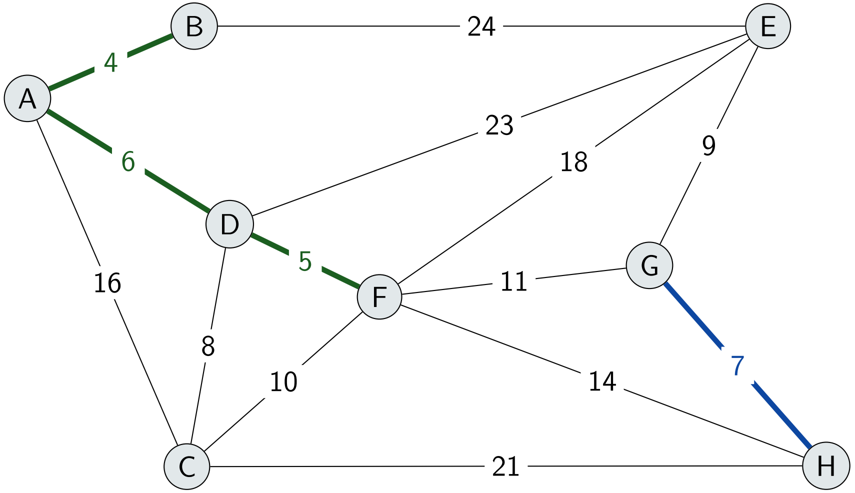
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. **A-D (6)**
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



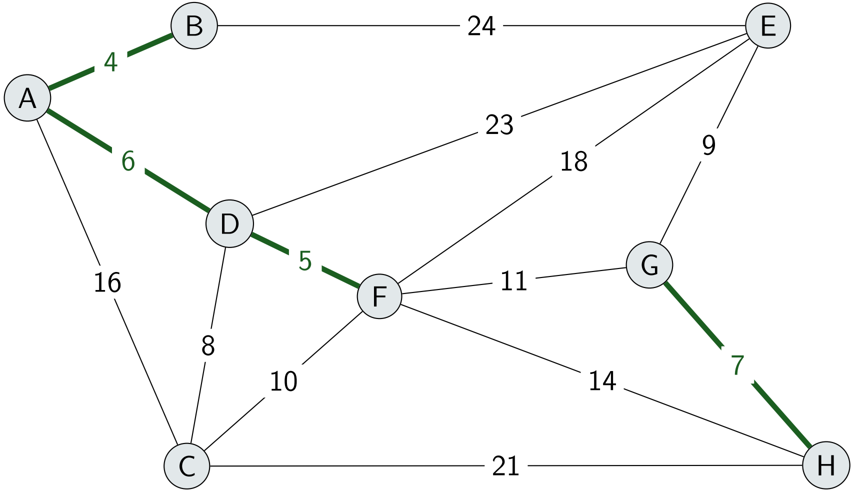
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. **G-H (7)**
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. ...



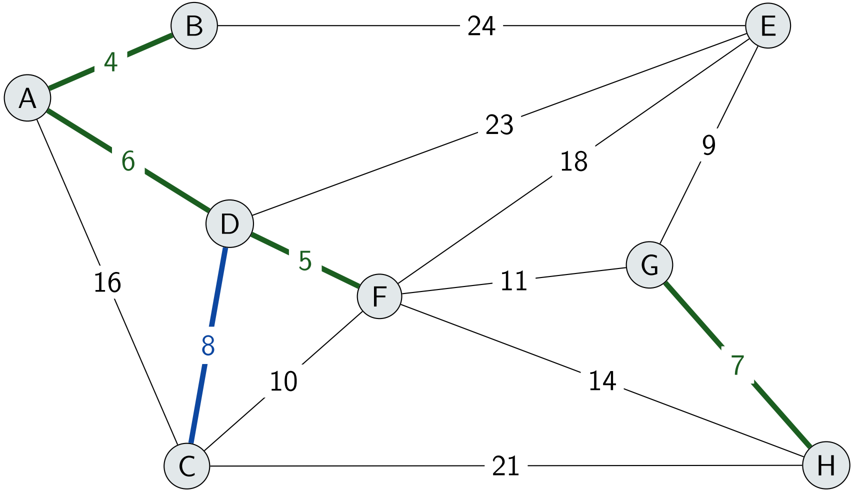
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. **G-H (7)**
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



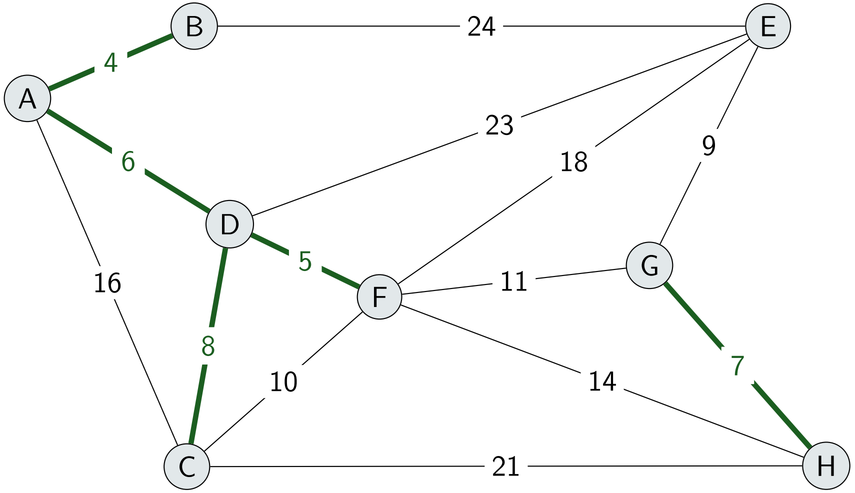
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. **C-D (8)**
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



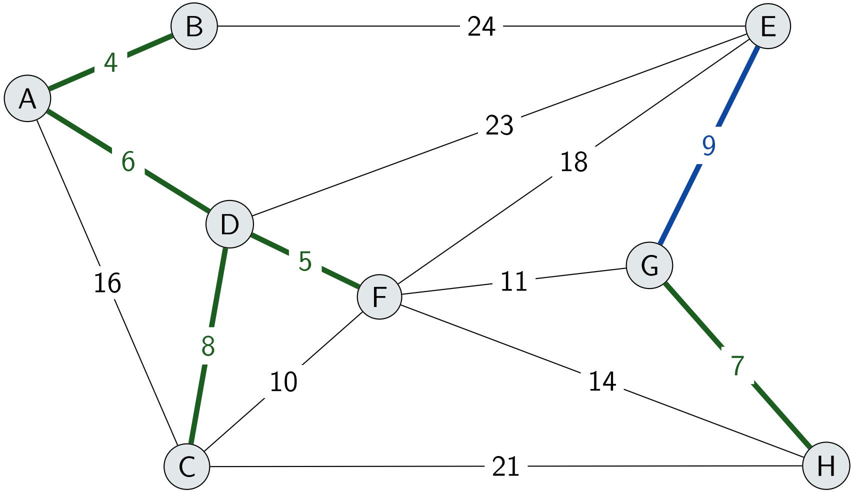
Kruskal’s algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. **C-D (8)**
6. E-G (9)
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



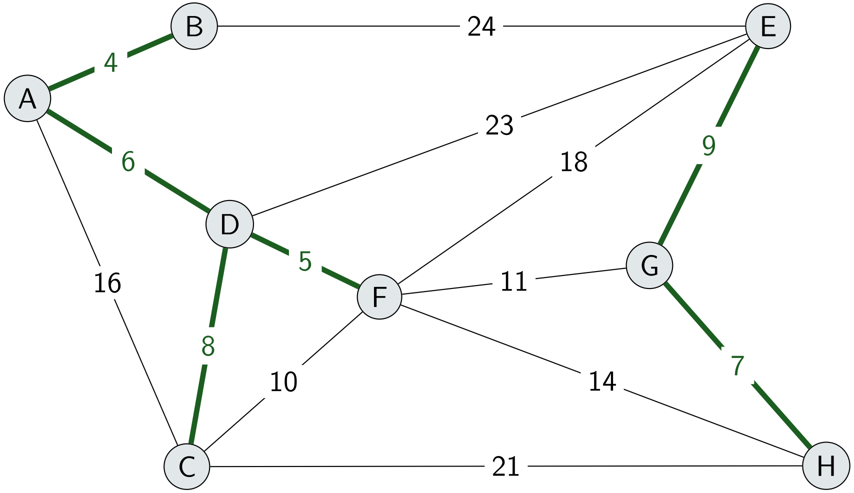
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. **E-G (9)**
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



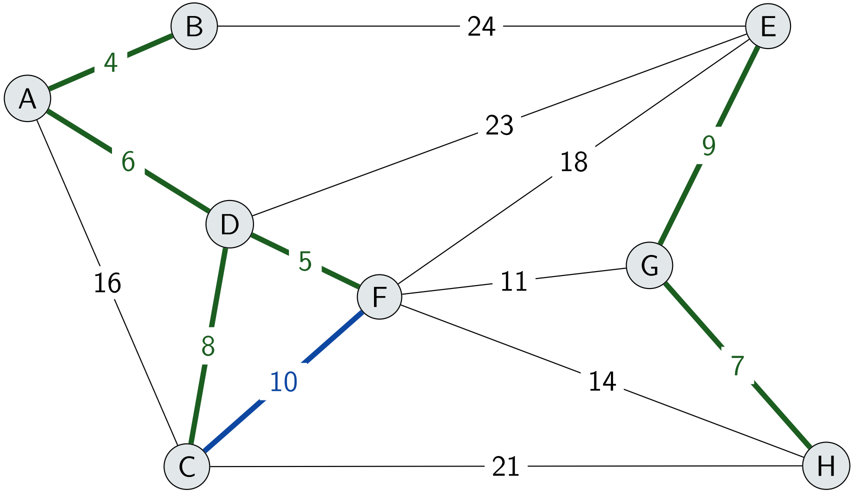
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. **E-G (9)**
7. C-F (10)
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



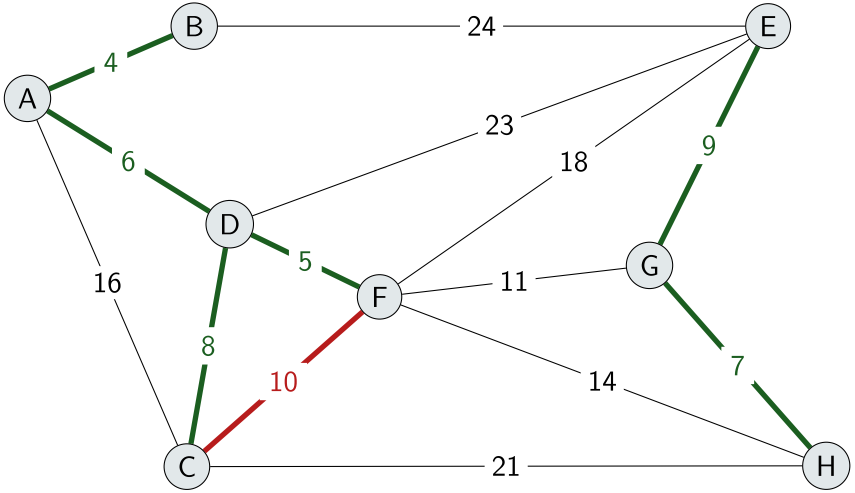
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. **C-F (10)**
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



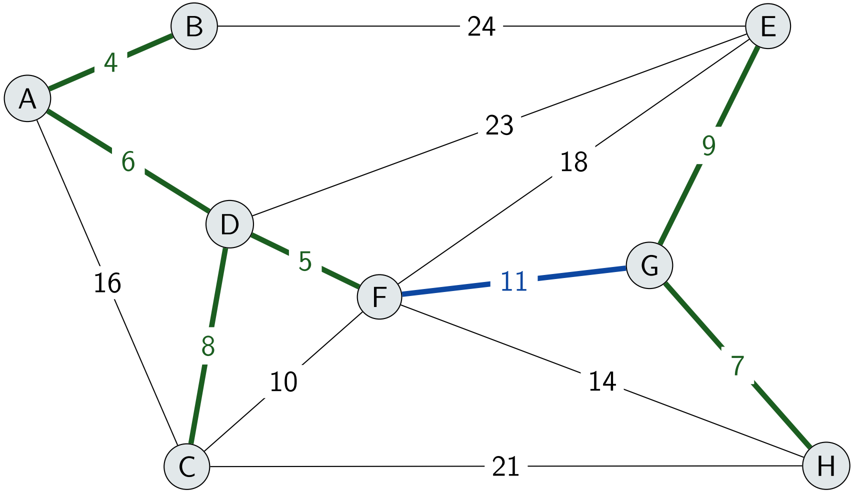
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. **C-F (10)**
8. F-G (11)
9. F-H (14)
10. A-C (16)
11. . . .



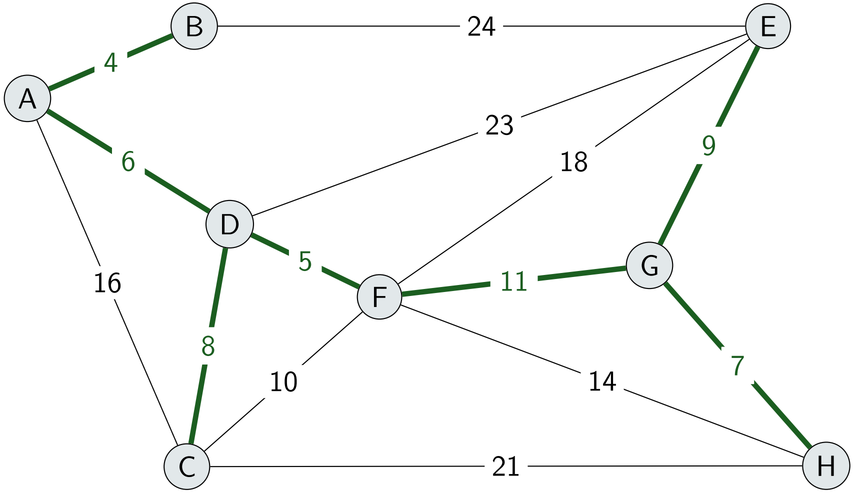
Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. **F-G (11)**
9. F-H (14)
10. A-C (16)
11. . . .



Kruskal's algorithm: example

1. A-B (4)
2. D-F (5)
3. A-D (6)
4. G-H (7)
5. C-D (8)
6. E-G (9)
7. C-F (10)
8. **F-G (11)**
9. F-H (14)
10. A-C (16)
11. . . .



Kruskal's algorithm: runtime

```
input  : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$.
output : minimum spanning tree $(V, F)$ of $G$

procedure Kruskal $(G, w)$
    sort edges by ascending by weight such that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$ ;
     // $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$
    $F \leftarrow \varnothing$ ;                                              // $\mathcal{O}(1)$
    for $i = 1$ to $m$ do                                                      // $m\times$
        $\{u, v\} \leftarrow e_i$ ;                                              // $\mathcal{O}(1)$
        if $u$ and $v$ are in different components of $(V, F)$ then           // $\mathcal{O}(n)$
            $F \leftarrow F \cup \{e_i\}$;                                      // $\mathcal{O}(1)$
```

Total: $\mathcal{O}(m \log n + mn) = \mathcal{O}(mn)$

Kruskal's algorithm: runtime

```
input  : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$.
output : minimum spanning tree $(V, F)$ of $G$

procedure Kruskal $(G, w)$
    sort edges by ascending by weight such that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$ ;
     // $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$
    $F \leftarrow \varnothing$ ;                                                    // $\mathcal{O}(1)$
    for $i = 1$ to $m$ do                                                           // $m\times$
        $\{u, v\} \leftarrow e_i$ ;                                                 // $\mathcal{O}(1)$
        if $u$ and $v$ are in different components of $(V, F)$ then                 // $\mathcal{O}(n)$
            $F \leftarrow F \cup \{e_i\}$;                                          // $\mathcal{O}(1)$
```

Can we do better? $\Rightarrow$ **Yes**, by using the **union-find** data-structure to keep track of the components of (V, F).

Kruskal's algorithm: improvement using union-find

Instead of calling BFS to check for every edge whether its two endpoints are in the same component, we use the union-find data-structure to keep track and check components.

Union-find: data-structure that can be used to keep track of a changing partition of a set (in our case, components of a graph).

Supported operations:

- `initUF`(V):
  Initializes and returns the union-find data-structure with each element in V being in its own set.
- `sameSet`(U, u, v):
  Returns true if and only if u and v are in the same set in U.
- `union`(U, u, v):
  Merges the set containing u with the set containing v.

Kruskal's Algorithm: implementation (with union-find)

input : graph $G = (V, E)$ and edge-weights $w \colon E \to \mathbb{Q}_{\geq 0}$.
output : minimum spanning tree (V, F) of G

```
procedure Kruskal $(G, w)$
    sort edges by ascending by weight such that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$;
    $F \leftarrow \varnothing$;
    $U \leftarrow$ initUF $(V)$;
    for $i = 1$ to $m$ do
        $\{u, v\} \leftarrow e_i$;
        if !sameSet $(U, u, v)$ then
            $F \leftarrow F \cup \{e_i\}$;
            union $(U, u, v)$;
```

Kruskal’s Algorithm: implementation (with union-find)

```
input  : graph $G = (V, E)$ and edge-weights $w\colon E \to \mathbb{Q}_{\geq 0}$.
output : minimum spanning tree $(V, F)$ of $G$

procedure Kruskal $(G, w)$
    sort edges by ascending by weight such that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$;
    $F \leftarrow \varnothing$;
    $U \leftarrow$ initUF $(V)$;
    for $i = 1$ to $m$ do
        $\{u, v\} \leftarrow e_i$;
        if !sameSet $(U, u, v)$ then
            $F \leftarrow F \cup \{e_i\}$;
            union $(U, u, v)$;
```

Runtime: Now dominated by the time to sort the edges, i. e., $\mathcal{O}(m \log n)$!

Kruskal vs. Prim (comparison)

Runtime of Kruskal:

- Union-find-operation is practically possible in constant time. Therefore, the second part of the algorithm is essentially linear (in m).
- The runtime is therefore dominated by sorting the edges, i. e., $\mathcal{O}(m \log n)$.

Runtime of Prim:

- If one uses a classical heap as a priority queue, the total runtime is $\mathcal{O}(m \log n)$.
- Can be improved to $\mathcal{O}(m + n \log n)$ if one uses a so-called **Fibonacci-Heap**.

Application in practise:

- For dense graphs ($m = \Theta(n^2)$) Prim's algorithm is more efficient
- For sparse graphs ($m = \Theta(n)$) Kruskal's algorithm is preferred.

UNIVERSITY OF LEEDS

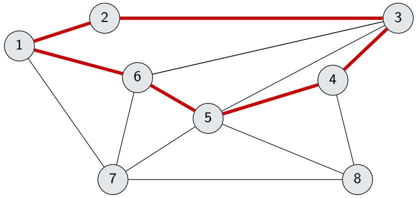
OCOM5102 – Algorithms

Correctness of Prim's and Kruskal's algorithms

Isolde Adler and Sebastian Ordyniak

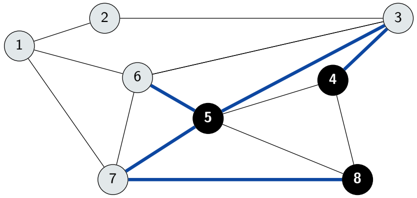
Cycles and Cuts

Cycle. Set of edges of the form: a-b, b-c, c-d, $\ldots$, y-z, z-a.



Cycle $C = (1, 2, 3, 4, 5, 6)$

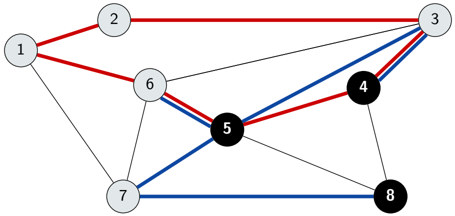
Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



Cut $S = \{4, 5, 8\}$
Cutset $D = \{5–6, 5–7, 3–4, 3–5, 7–8\}$

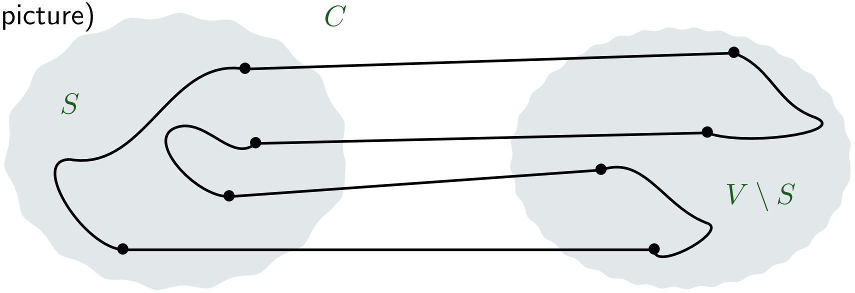
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle $C = (1, 2, 3, 4, 5, 6)$
Cutset $D = \{3\text{-}4, 3\text{-}5, 5\text{-}6, 5\text{-}7, 7\text{-}8\}$
Intersection 3-4 and 5-6

Pf. (by picture)

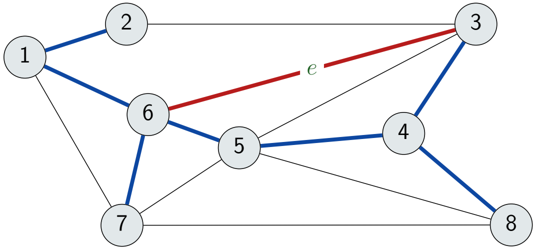


Exchange Property for Spanning Trees

Let T be a spanning tree in a graph $G = (V, E)$ and let $e \in E$.

Observation: the graph $T + \{e\}$, i. e., the graph obtained after adding e to T contains a cycle C that contains e.

Proof: because the endpoints of e are connected by a path in T



$C = (6, 3, 4, 5)$

Exchange Property for Spanning Trees

Let T be a spanning tree in a graph $G = (V, E)$ and let $e \in E$.

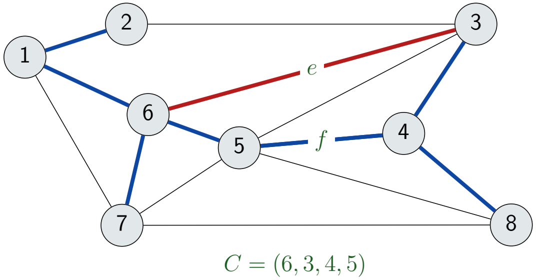
Observation: the graph $T + \{e\}$, i. e., the graph obtained after adding e to T contains a cycle C that contains e.

Proof: because the endpoints of e are connected by a path in T

Observation: $T' = T + \{e\} - \{f\}$ is also a spanning tree for every edge f on C.

Proof:

- Two vertices that were connected through f in T are now connected through e in T'.
- If there were a cycle in T' using e then there was a cycle in T using f.



Choosing the right MST

Definition:

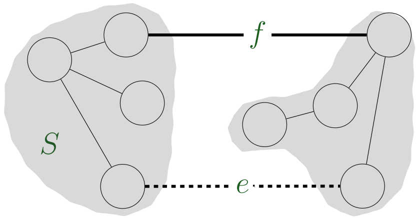
We say that an MST T **agrees with Prim's algorithm until step $\boldsymbol{i}$** if T contains the first i edges chosen by Prim's algorithm.

Definition:

We say that an MST T **agrees with Kruskal's algorithm until step $\boldsymbol{i}$** if T contains the first i edges chosen by Kruskal's algorithm.

Prim's algorithm: correctness proof (exchange argument)

- Let i be the largest integer such that there is an MST T_O that agrees with Prim's algorithm until step i.
- Without loss of generality, we can assume that $i < n - 1$ since otherwise Prim's algorithm outputs T_O and there is nothing to show.
- Let e be the edge chosen at step $i + 1$ of Prim's algorithm and let S be the set of vertices before e is chosen.
- Adding e to T_O creates a cycle C in $T_O + \{e\}$.
- Edge e is both in the cycle C and in the cutset D of S $\Rightarrow$ there exists another edge, say f, that is in both C and D.
- $T'_O = T_O + \{e\} - \{f\}$ is also an ST and because $w(e) \leq w(f)$, T'_O is also a MST.
- However, T'_O agrees with Prim's algorithm until step $i + 1$, contradicting our choice of i.



Kruskal's algorithm: correctness proof (exchange argument)

- Let T_K be the ST computed by Kruskal's algorithm.
- Let i be the largest integer such that there is an MST T_O that agrees with Kruskal's algorithm until step i.
- Without loss of generality, we can assume that $i < n - 1$ since otherwise $T_K = T_O$ and there is nothing to show.
- Let e be the edge chosen at step $i + 1$ by Krukal's algorithm.
- Adding e to T_O creates a cycle C in $T_O + \{e\}$.
- Since C is not in T_K but e is in T_K, there is an edge f in C that is not in T_K.
- Since f was not chosen by Kruskal's algorithm and f makes no cycle with the first i edges (because f is in T_O), it holds that $w(e) \leq w(f)$.
- Therefore, $T'_O = T_O + \{e\} - \{f\}$ is also an MST.
- However, T'_O agrees with Kruskal's algorithm until step $i + 1$, contradicting our choice of i.

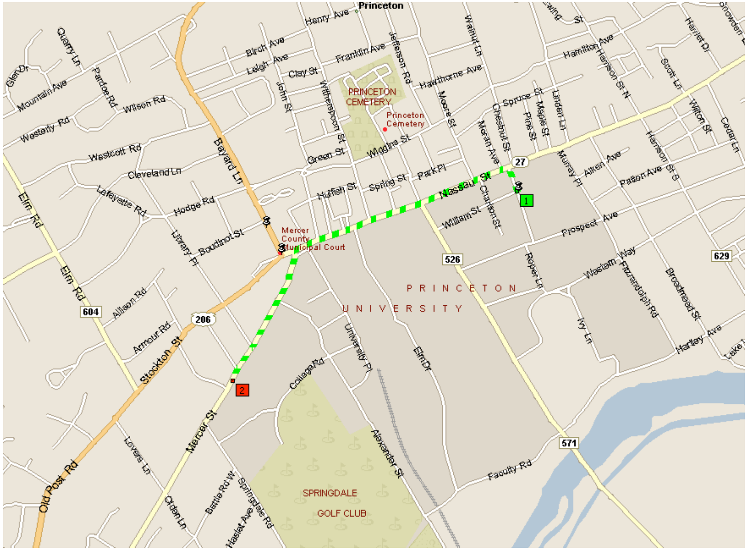
UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Greedy algorithms: Dijkstra's algorithm

Isolde Adler and Sebastian Ordyniak

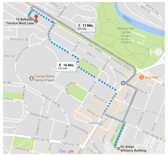
Shortest path in a weighted directed graph

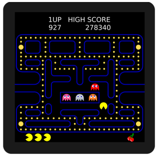


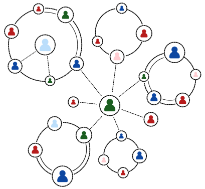
Shortest path from the computer science institute in Princeton to Einstein's house.

Shortest paths: applications

- Navigation
- Pathfinding
- Routing messages in a network
- Social network analysis
- ...







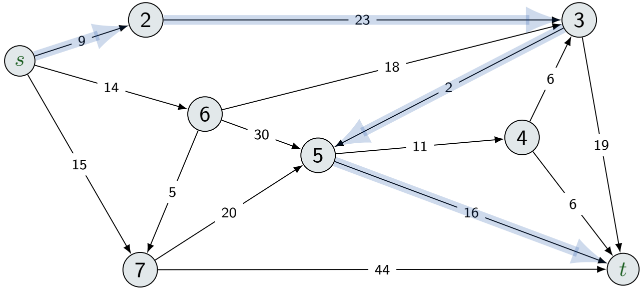
Shortest path problem

Given:

- a directed graph $G = (V, E)$,
- a length (weight/cost) function $w\colon E \to \mathbb{Q}_{\geq 0}$ and
- a source (initial) vertex s.

Problem: Find a shortest directed path from s to every other vertex in D.

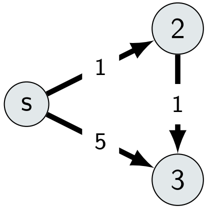
□ *shortest path ≈ path of minimum length, measured as the sum of the lengths of all arcs on the path.*



Total length of the path s-2-3-5-t:
$9 + 23 + 2 + 16$
$= 50$.

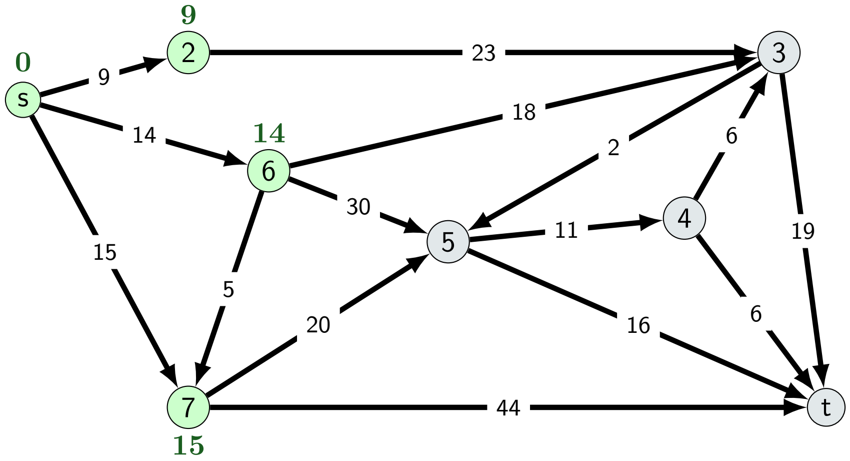
Dijkstra's algorithm: main idea

- Works similar to BFS or Prim's:
  - Grow a set S of nodes for which you have already computed a shortest path from s.
  - Always pick some vertex next that is closest to s among all nodes outside of S.
- However, because edges can have different lengths, finding the next vertex becomes slightly more complicated.

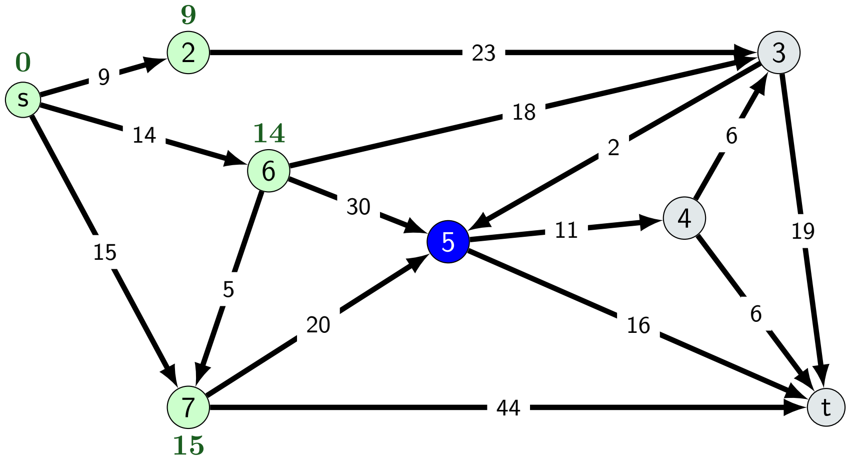


An example where BFS does not work, but we need Dijkstra!

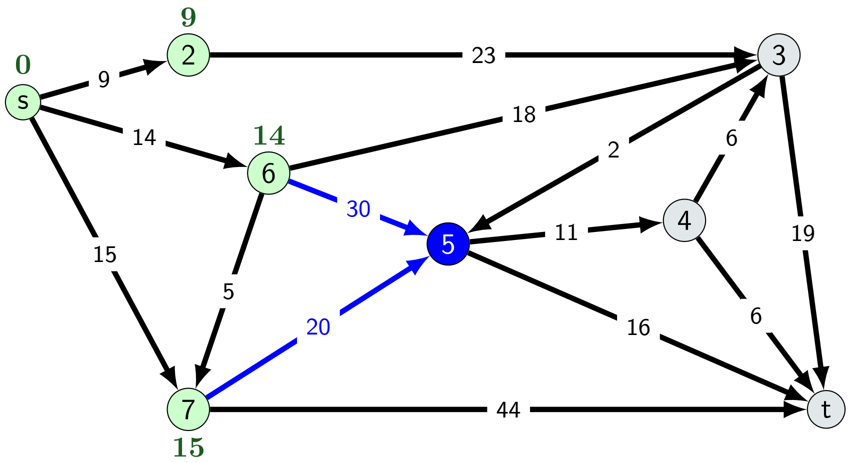
Dijkstra's algorithm: growing of the set S



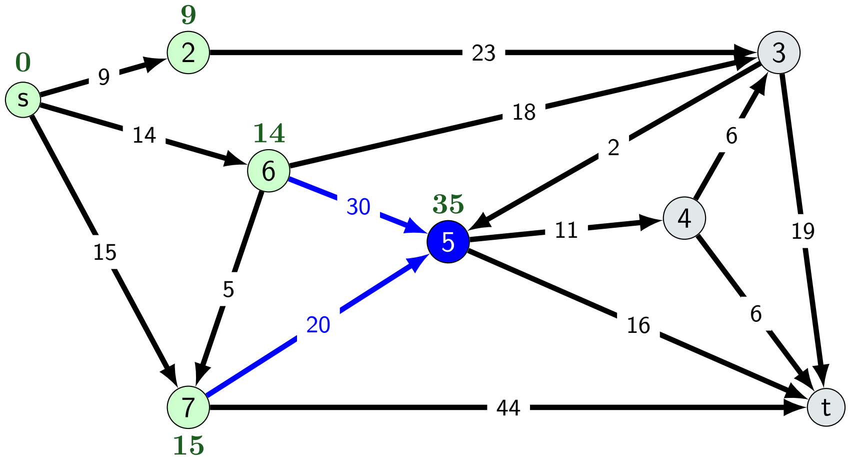
Dijkstra's algorithm: growing of the set S



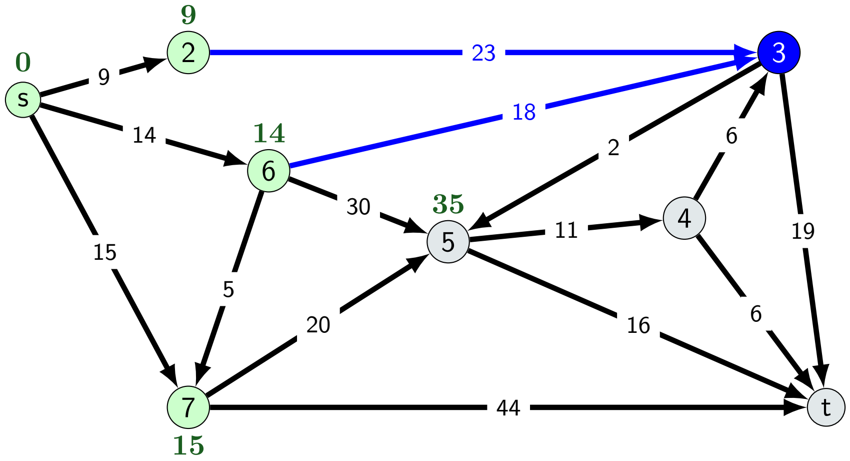
Dijkstra's algorithm: growing of the set S



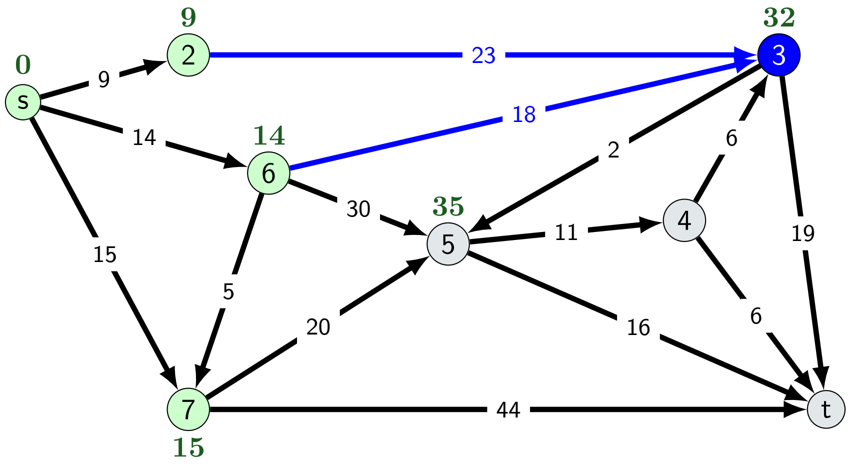
Dijkstra's algorithm: growing of the set S



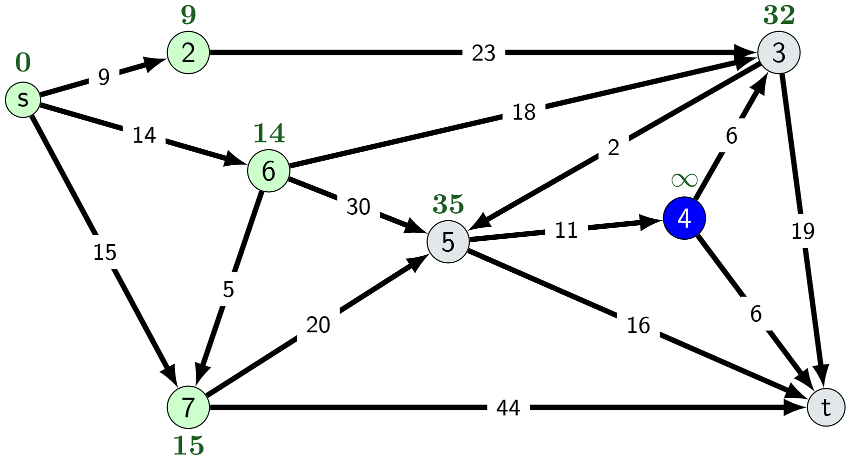
Dijkstra's algorithm: growing of the set S



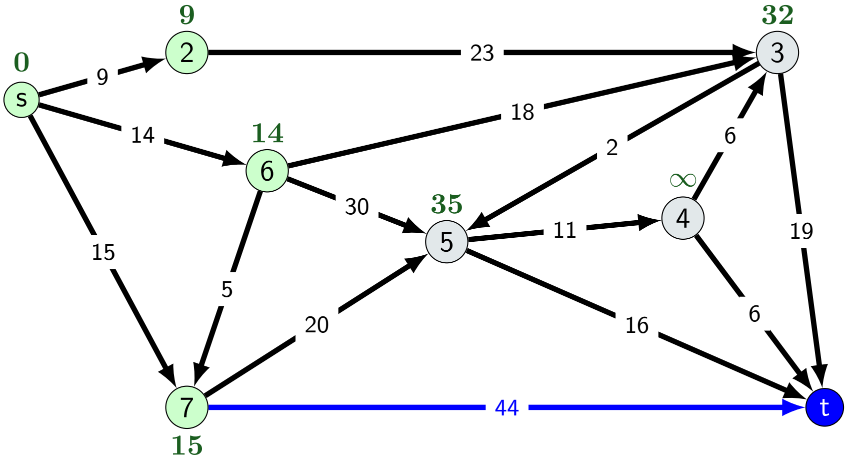
Dijkstra's algorithm: growing of the set S



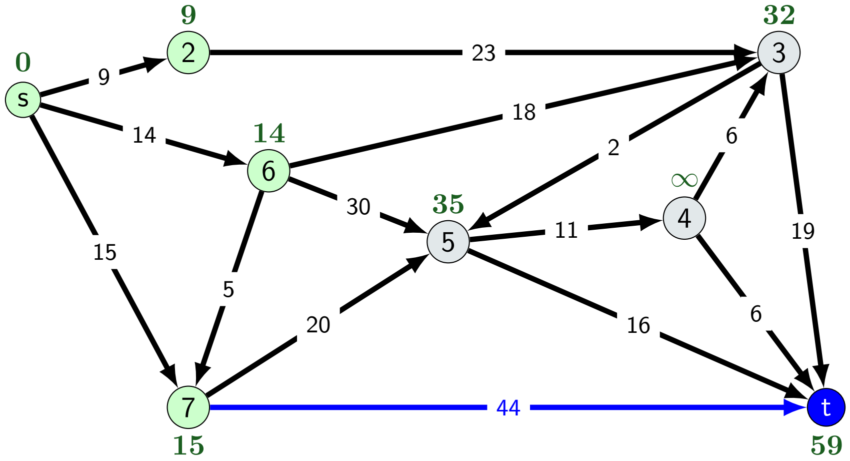
Dijkstra's algorithm: growing of the set S



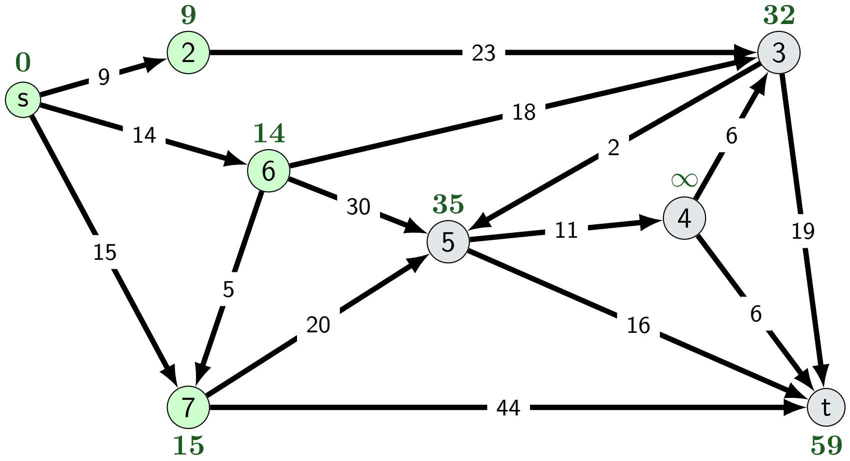
Dijkstra's algorithm: growing of the set S



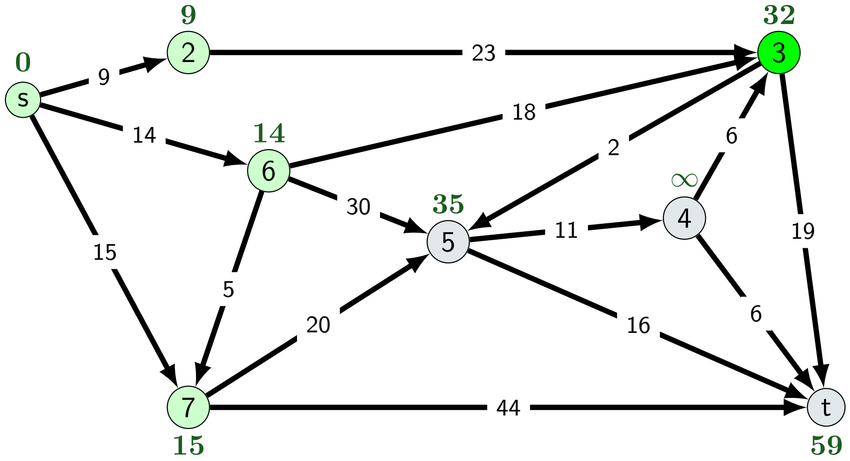
Dijkstra's algorithm: growing of the set S



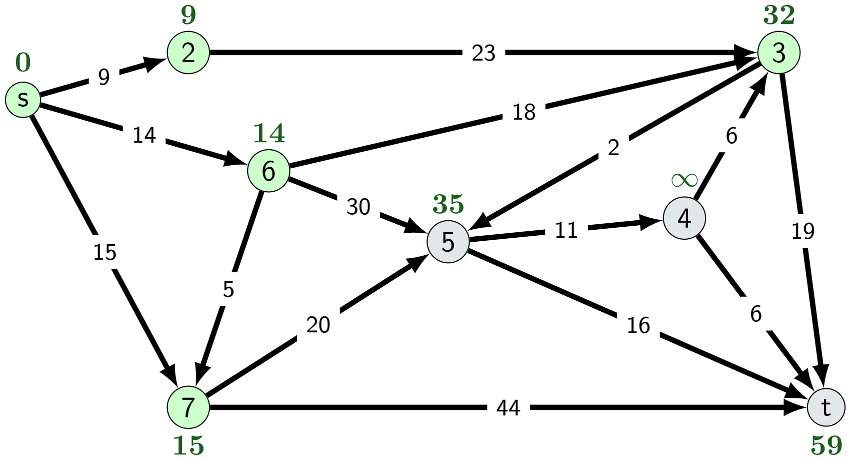
Dijkstra's algorithm: growing of the set S



Dijkstra's algorithm: growing of the set S



Dijkstra's algorithm: growing of the set S



Dijkstra's algorithm: outline

- Iteratively grow a set S of **visited vertices**, for which we already calculated the length $l(u)$ of a shortest s-u-path.
- Initialize $S = \{s\}$, $l(s) = 0$.
- Choose a vertex v that minimizes the length of any shortest $s - v$ path that uses only vertices in $S \cup \{v\}$, i. e., v minimizes the following value:

$$\min_{u \in S \wedge v \in N^+(u)} l(u) + w(u, v).$$

- Add v to S, set $l(v) = \min_{u \in S \wedge v \in N^+(u)} l(u) + w(u, v)$ and continue with the previous step.

Dijkstra's algorithm: improvement

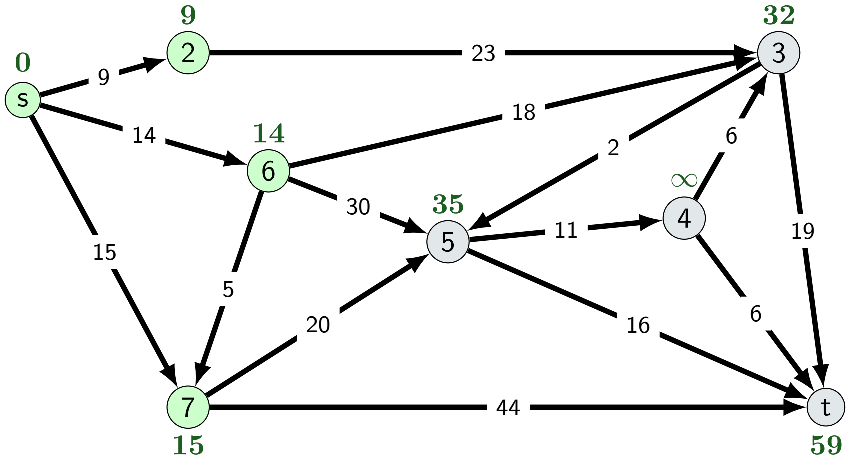
To avoid having to recalculate $\min\limits_{u \in S \wedge v \in N^+(u)} l(u) + w(u,v)$ at each step of the algorithm and for all vertices outside of S, one can remember the previous values and only update values if a new vertex is added to S, i. e.:

- One initially sets $S = \varnothing$, $l(s) = 0$, and $l(v) = \infty$ for every $v \in V \setminus \{s\}$.
- Every time a vertex v is added to S one sets

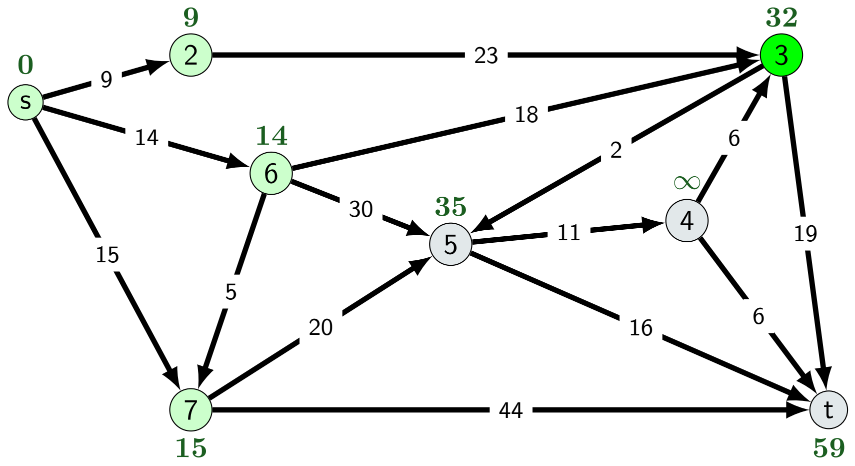
  $$l(x) = \min\{l(x), l(v) + w(v,x)\}$$

  for every vertex $x \in N^+(v) \setminus S$.
- The next vertex to be added to S is then always a vertex that minimizes $l(v)$ among all vertices outside of S.

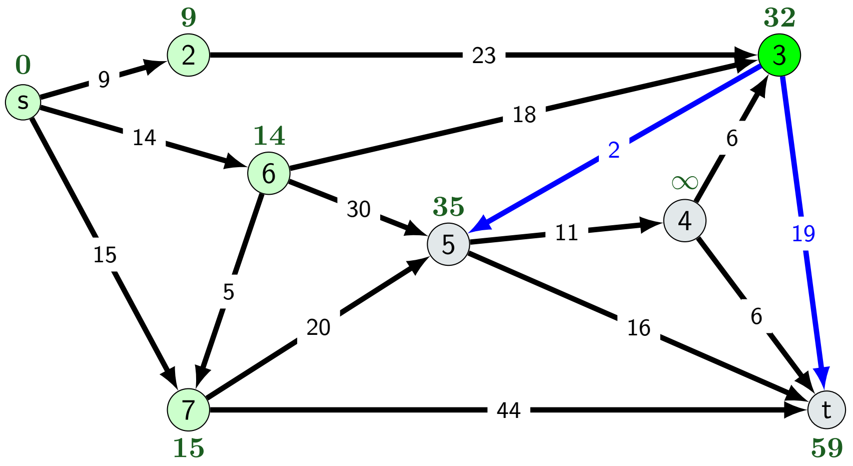
Dijkstra's algorithm: example (updating lengths)



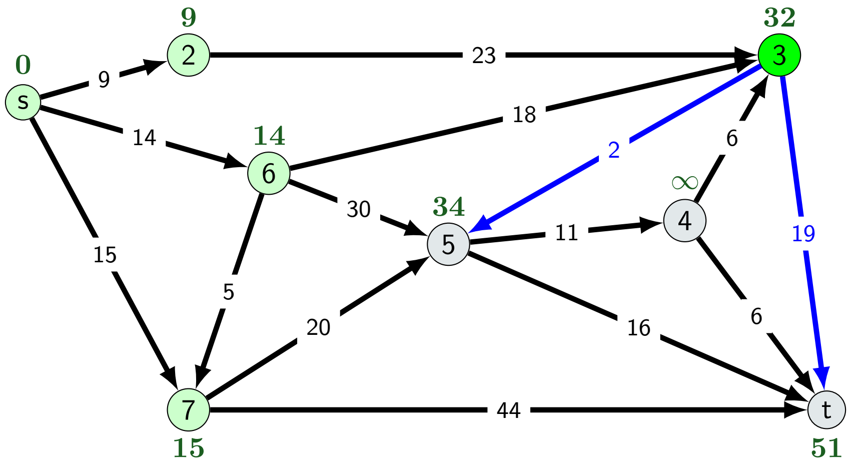
Dijkstra's algorithm: example (updating lengths)



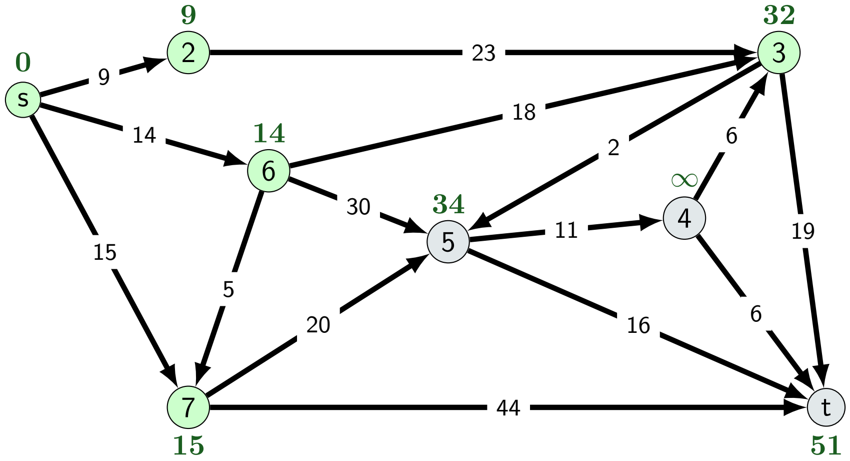
Dijkstra's algorithm: example (updating lengths)



Dijkstra's algorithm: example (updating lengths)



Dijkstra's algorithm: example (updating lengths)



Dijkstra's algorithm: implementation

We will consider two datastructues for the computation of the set S:

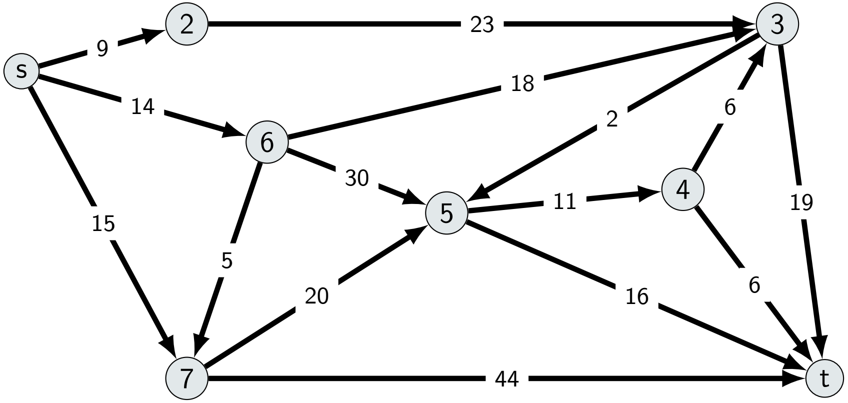
- A **simply-linked list**.
- A **priority queue** for the nodes outside of S ordered w.r.t. to the length l. An entry in the queue consists of the node-index and its currently associated length.

Dijkstra's algorithm: pseudo-code (linked-list L)

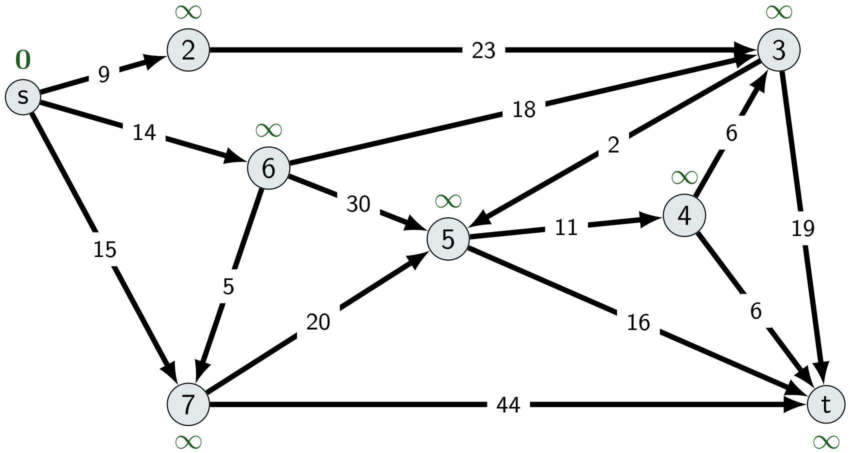
```
input  : graph $D = (V, E)$, $w : E \to \mathbb{Q}_{\geq 0}$ and start vertex $s$.
output : $l[v]$, the distance from $s$ to $v$, shortest path tree $(V, F)$

procedure Dijkstra$(D, w, s)$
    for $v \in V \setminus \{s\}$ do  visited$[v]$ $\leftarrow$false; $l[v] \leftarrow \infty$;
    ;
    $L \leftarrow V \setminus \{s\}$; $F \leftarrow \varnothing$; visited$[s]$ $\leftarrow$true; $l[s] \leftarrow 0$;
    for $v \in N^+(s)$ do $l[v] \leftarrow w(s, v)$; $p[v] \leftarrow s$;
    ;
    while $L \neq \varnothing$ do
        choose $u \in L$ with smallest $l[u]$;
        $L \leftarrow L \setminus \{u\}$; $F \leftarrow F \cup \{(p[u], u)\}$; visited$[u]$ $\leftarrow$true;
        for $v \in N^+(u)$ do
            if !visited$[v]$ and $l[v] > l[u] + w(u, v)$ then
                $l[v] \leftarrow l[u] + w(u, v)$;
                $p[v] \leftarrow u$;
```

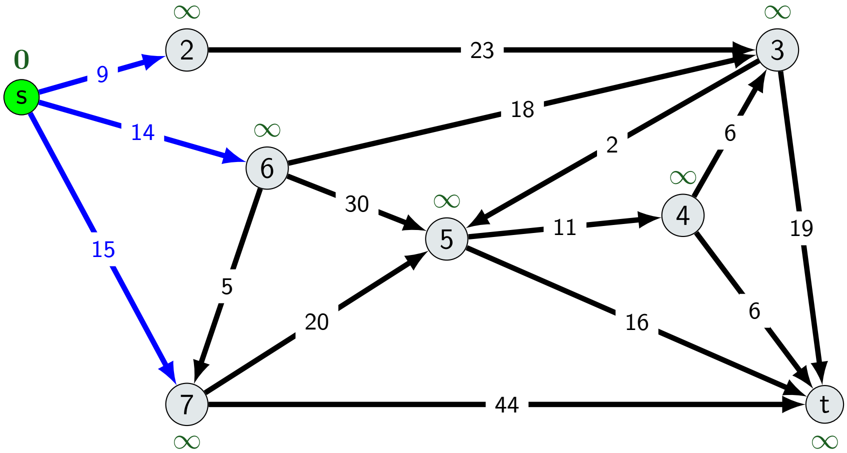
Dijkstra's algorithm: example



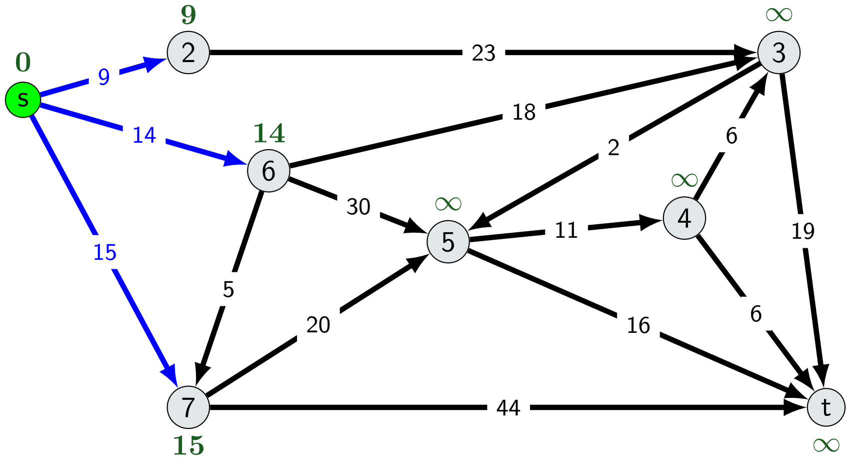
Dijkstra's algorithm: example



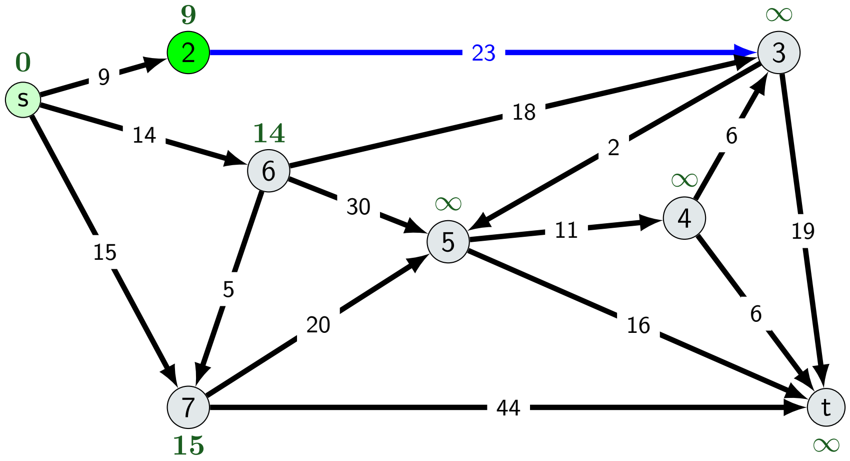
Dijkstra's algorithm: example



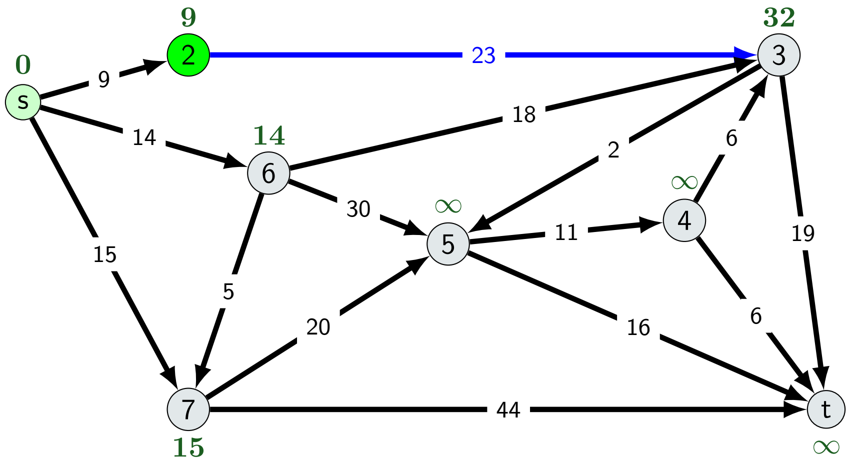
Dijkstra's algorithm: example



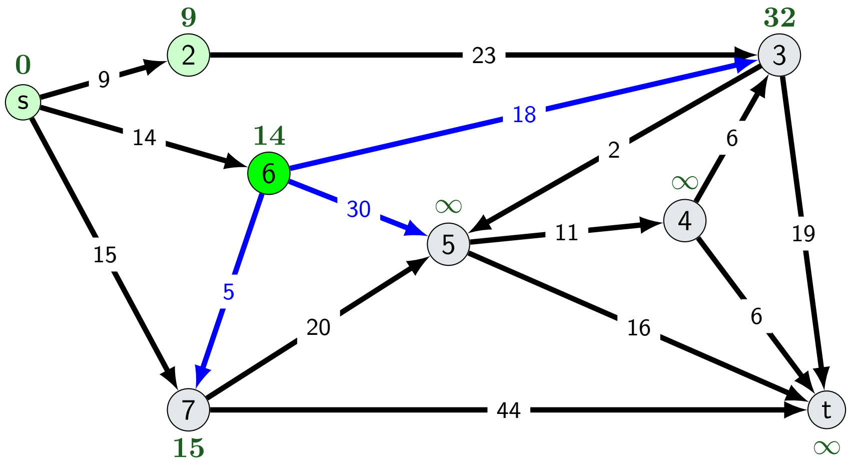
Dijkstra's algorithm: example



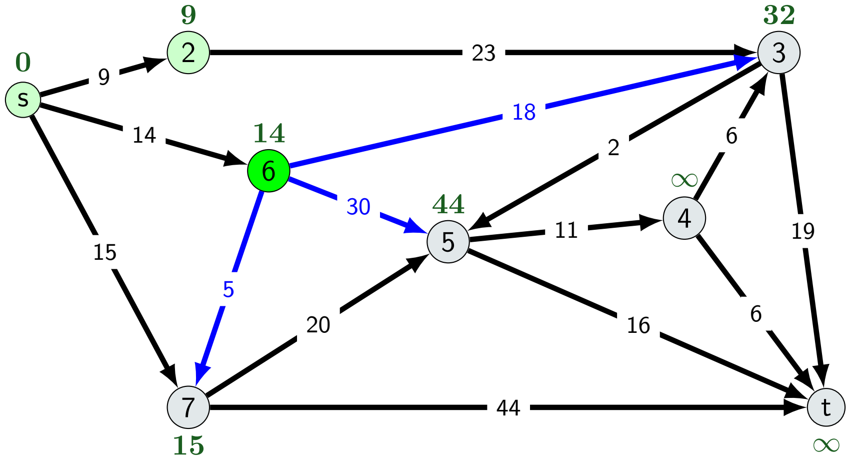
Dijkstra's algorithm: example



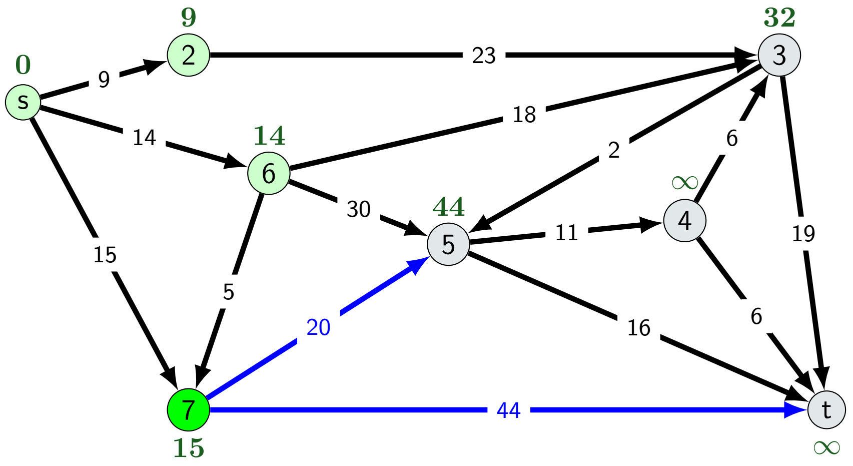
Dijkstra’s algorithm: example



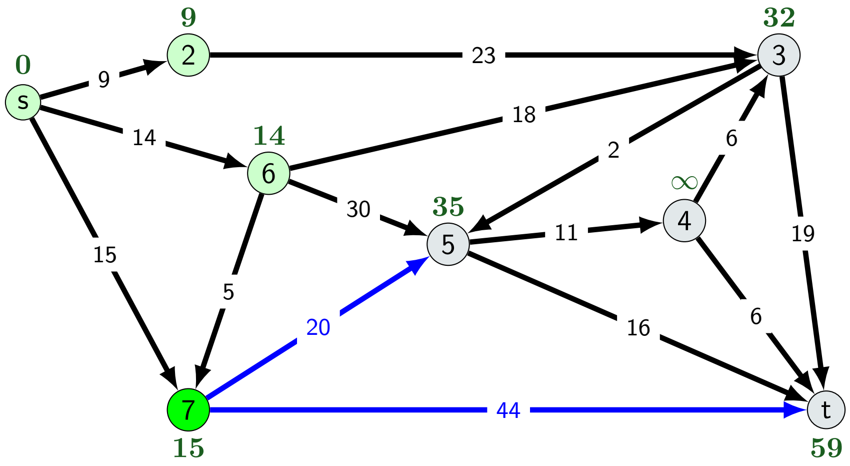
Dijkstra's algorithm: example



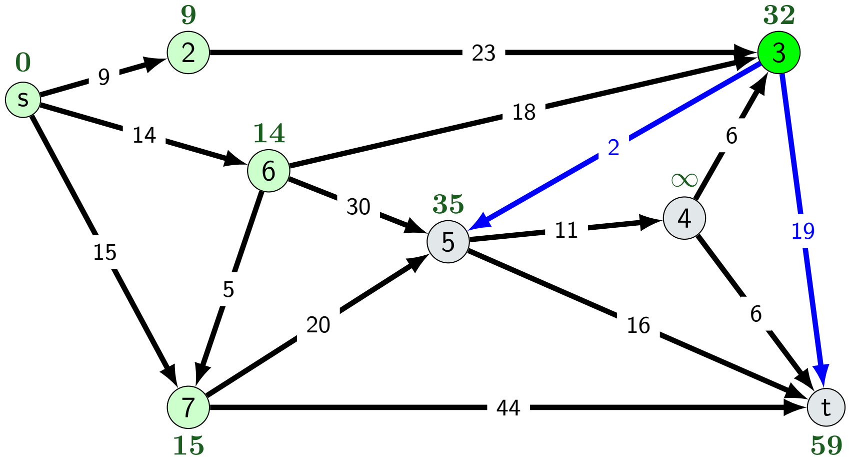
Dijkstra's algorithm: example



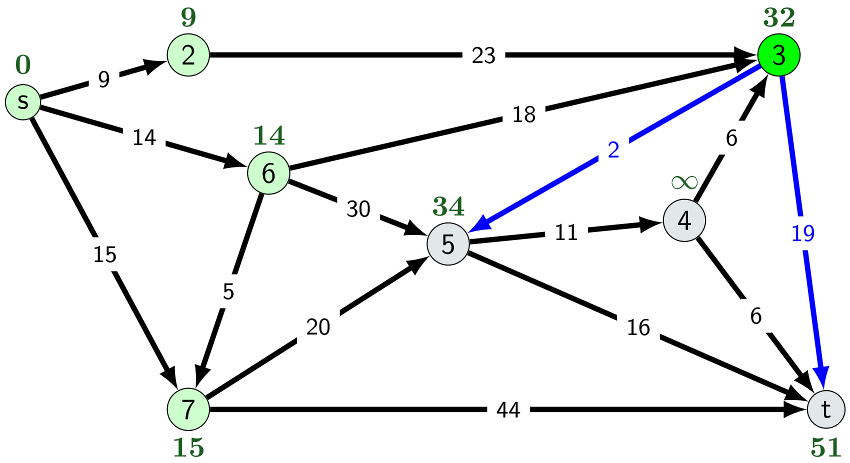
Dijkstra's algorithm: example



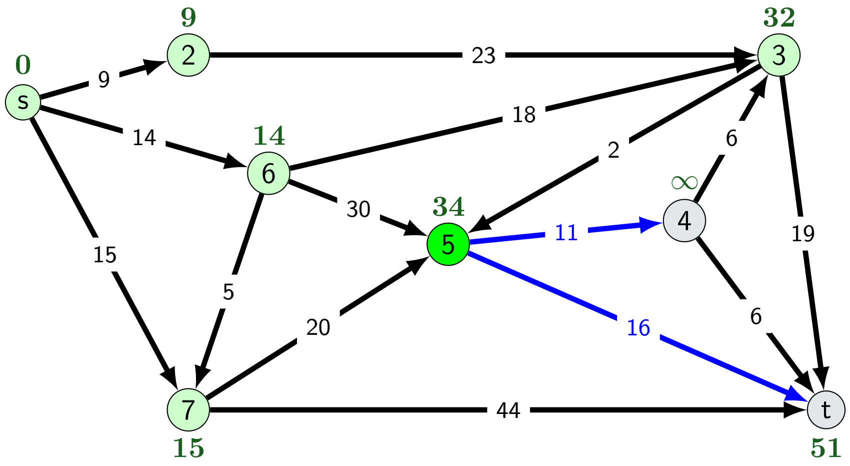
Dijkstra's algorithm: example



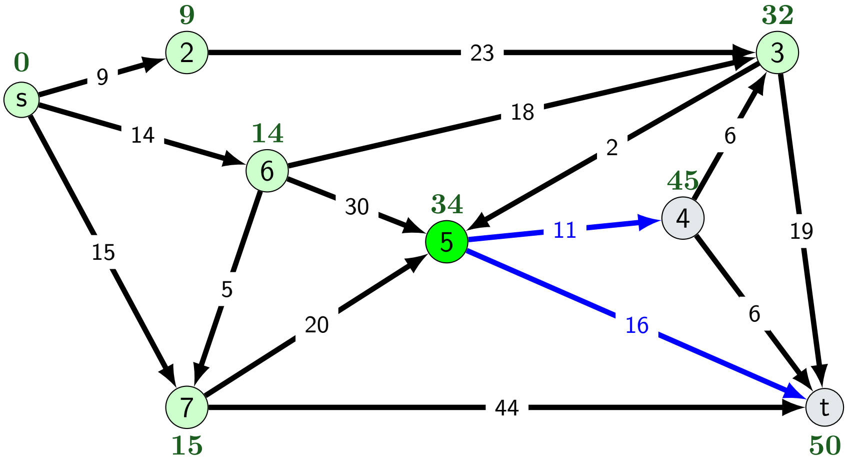
Dijkstra's algorithm: example



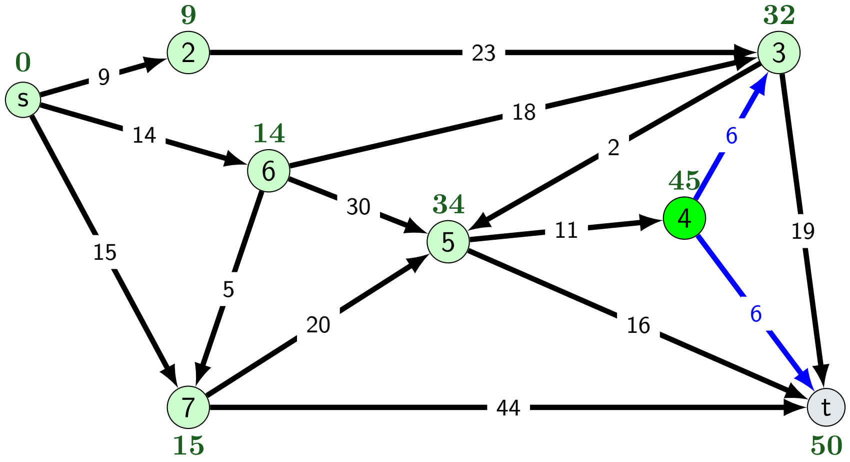
Dijkstra's algorithm: example



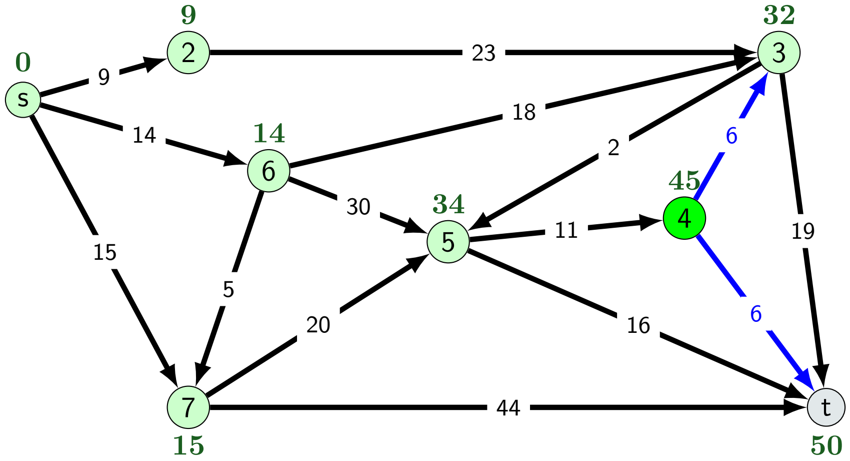
Dijkstra's algorithm: example



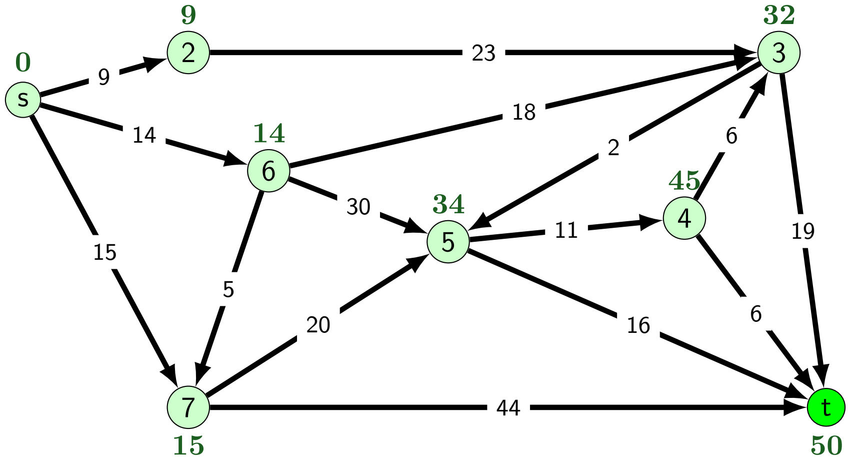
Dijkstra's algorithm: example



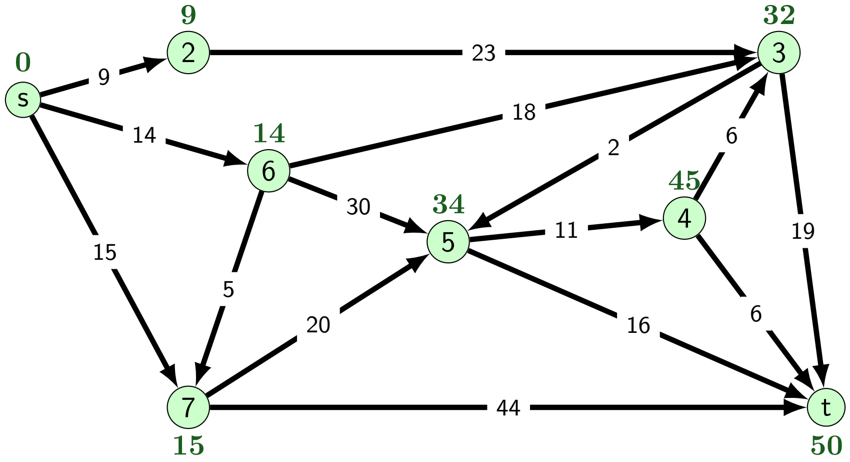
Dijkstra's algorithm: example



Dijkstra's algorithm: example



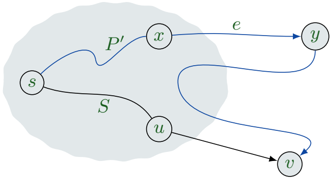
Dijkstra's algorithm: example



Dijkstra's algorithm: correctness

Invariant: $l(u)$ is the lengths of a shortest s-u path for every $u \in S$.
Proof: (Induction on $|S|$)
Induction start: $|S| = 1$ trivial.
Induction step: Assume the claim holds for $|S| = k \geq 1$.

- Let v be the next vertex added to S and let $\{u, v\}$ be the chosen edge.
- A shortest s-u path plus (u, v) is a s-v path of lengths $l(v)$.
- Let P be an arbitrary s-v path. We will show that P is not shorter than $l(v)$.
- Let $e = \{x, y\}$ be the first arc on P that leaves S and let P' be the part of P from s to x.
- Then P is already too long, when it leaves S.



$$\text{length}(P) \geq \text{length}(P') + w(e) \geq l(x) + w(e) \geq l(y) \geq l(v)$$

🟩 *non-negative lengths* 🟦 *induction hypothesis* 🟥 *definition of $l(y)$*
⬜ *Dijkstra's algorithm chooses v instead of y*

Dijkstra’s algorithm: running time (linked-list L)

```
procedure Dijkstra$(D, w, s)$
    for $v \in V \setminus \{s\}$ do
        $\ldots$;                                                  // $\mathcal{O}(n)$
    $L \leftarrow V \setminus \{s\}$; $\ldots$; $l[s] \leftarrow 0$;      // $\mathcal{O}(1)$
    for $v \in N^+(s)$ do
        $\ldots$;                                                  // $\mathcal{O}(n)$
    while $L \neq \varnothing$ do
        ;                                                          // $n \times$
        choose $u \in L$ with smallest $l[u]$;                     // $\mathcal{O}(n)$
        $\ldots$;                                                  // $\mathcal{O}(1)$
        for $v \in N^+(u)$ do
            $\ldots$;                                              // $\mathcal{O}(\deg^+(u))$
```

Total: $\mathcal{O}\big(n + 1 + \sum_{u \in V} (n + 1 + \deg^+(u))\big) = \mathcal{O}(n^2)$

Improvement using priority queue

The running time of Dijkstra's algorithm can be significantly reduced by using a priority queue to maintain the distances of the vertices outside of S.

Recap: Operations supported and required from priority queue:

- `initializePriorityQueue`(U, l):
  Initializes and returns a priority queue with the elements in the set U using the priorities given by the array l, i. e., $l[u]$ is the priority of $u \in U$.

- `extractMin(`Q`)`:
  Removes and returns an element in Q with minimum priority.

- `decreaseKey(`Q,s,p`)`:
  Sets the priority of the element u in Q to p.

Dijkstra's algorithm: running time (priority queue Q)

```
procedure Dijkstra($(D, w, s)$)
    ...
    $Q \leftarrow$ initializePriorityQueue$(V \setminus \{s\}, l)$;                         // $1 \times$
    while $Q \neq \emptyset$ do
        $u \leftarrow$ extractMin$(Q)$;                                                    // $n \times$
        ...
        for $v \in N^+(u)$ do
            if !visited$[v]$ and $l[v] > l[u] + w(u, v)$ then
                decreaseKey$(Q, v, l[u] + w(u, v))$;                                       // $\mathcal{O}(m) \times$
                ...
```

Total:

$$\mathcal{O}(1 \times \text{rt}(\texttt{initializePriorityQueue}) + n \times \text{rt}(\texttt{extractMin}) + m \times \text{rt}(\texttt{decreaseKey}))$$

☐ $\text{rt}(P) \approx$ running time of procedure/function P

Dijkstra’s algorithm: complexity

Let $n = |V|$ and $m = |E|$. The total running time of the algorithm depends on the implementation of the priority queue Q.

Operation		Queue Implementation		
Name	#	List	Minimum Heap	Fibonacci Heap[1]
`decreaseKey`	m	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
`extractMin`	n	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`initializePriorityQueue`	1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Total		$\mathcal{O}(n^2 + m)$ $= \mathcal{O}(n^2)$	$\mathcal{O}((n+m)\log n)$	$\mathcal{O}(n \log n + m)$

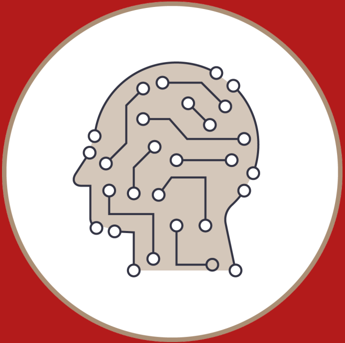
[1] amortized complexity

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

A-star search algorithm

Isolde Adler and Sebastian Ordyniak



A^* search algorithm: motivation

- Goal: modify Dijkstra's algorithm to make it faster **in many real-world scenarios**

A^* search algorithm: motivation

- Goal: modify Dijkstra's algorithm to make it faster **in many real-world scenarios**
- Variant: given start node s and target node t, find shortest s-t-path if exists

A^* search algorithm: motivation

- Goal: modify Dijkstra's algorithm to make it faster **in many real-world scenarios**
- Variant: given start node s and target node t, find shortest s-t-path if exists
- A^* search algorithm $\approx$ 'Dijkstra $+$ heuristic'.

Heuristic

Heuristic

- A speculation, estimation, or educated guess that guides the search for a solution to a problem.
- For example, spam filters flag a message as probable spam if it contains certain words, is sent to many people, has certain attachments etc.
- Hope that it works well in many cases in practice (no guarantees).

A^* search algorithm: motivation

- Goal: modify Dijkstra's algorithm to make it faster **in many real-world scenarios**
- Variant: given start node s and target node t, find shortest s-t-path if exists
- A^* search algorithm $\approx$ 'Dijkstra $+$ heuristic'

A^* search algorithm: motivation

- Goal: modify Dijkstra's algorithm to make it faster **in many real-world scenarios**
- Variant: given start node s and target node t, find shortest s-t-path if exists
- A^* search algorithm $\approx$ 'Dijkstra $+$ heuristic'
- Heuristic here: good guess in which direction to explore
  - Recall: in each step, Dijkstra's algorithm finds a new vertex v of **shortest distance to the start vertex** via the set S of visited vertices, and adds v to S

A^* search algorithm: motivation

- Goal: modify Dijkstra's algorithm to make it faster **in many real-world scenarios**
- Variant: given start node s and target node t, find shortest s-t-path if exists
- A^* search algorithm $\approx$ 'Dijkstra $+$ heuristic'
- Heuristic here: good guess in which direction to explore
  - Recall: in each step, Dijkstra's algorithm finds a new vertex v of **shortest distance to the start vertex** via the set S of visited vertices, and adds v to S
- A^* idea: replace **shortest distance to the start vertex** by:
  shortest distance to the start vertex $+$ estimated distance from v to t.

A^* search algorithm: motivation

- Goal: modify Dijkstra's algorithm to make it faster **in many real-world scenarios**
- Variant: given start node s and target node t, find shortest s-t-path if exists
- A^* search algorithm $\approx$ 'Dijkstra $+$ heuristic'
- Heuristic here: good guess in which direction to explore
  - Recall: in each step, Dijkstra's algorithm finds a new vertex v of **shortest distance to the start vertex** via the set S of visited vertices, and adds v to S
- A^* idea: replace **shortest distance to the start vertex** by:
  shortest distance to the start vertex $+$ estimated distance from v to t.

Applications

- At the core of many modern route planners / sat navs
- Used for searching in virtual reality / computer games / chip design

Heuristic functions

Given: graph $D = (V, E)$, $w : E \to \mathbb{Q}_{\geq 0}$.

- Suppose we are at node $v \in V$ and we want to find the shortest path to target t. A **heuristic function** is a function $h \colon V(D) \to \mathbb{Q}_{\geq 0}$.
- Intuitively, the value $h(v)$ should give us an 'estimate' of how far v is away from t.
- If I am in Keswick, cycling to Leeds, then $h(\text{Keswick})$ tells me approximately how much longer I have to cycle.

Heuristic functions

Given: graph $D = (V, E)$, $w : E \to \mathbb{Q}_{\geq 0}$.

- Suppose we are at node $v \in V$ and we want to find the shortest path to target t. A **heuristic function** is a function $h \colon V(D) \to \mathbb{Q}_{\geq 0}$.
- Intuitively, the value $h(v)$ should give us an 'estimate' of how far v is away from t.
- If I am in Keswick, cycling to Leeds, then $h(\text{Keswick})$ tells me approximately how much longer I have to cycle.

Examples

- The function $h \colon V \to \mathbb{Q}_{\geq 0}$ with $h(v) = 0$ for all nodes $v \in V$
- If our nodes are in the Euclidean (i. e. 2-dimensional) plain, then the straight-line distance, given by
$$h(v) = \sqrt{(v_x - t_x)^2 + (v_y - t_y)^2}$$
for all $v \in V$ is a heuristic function.

Heuristic functions

Given: graph $D = (V, E)$, $w : E \to \mathbb{Q}_{\geq 0}$.

- Suppose we are at node $v \in V$ and we want to find the shortest path to target t. A **heuristic function** is a function $h\colon V(D) \to \mathbb{Q}_{\geq 0}$.
- Intuitively, the value $h(v)$ should give us an 'estimate' of how far v is away from t.
- If I am in Keswick, cycling to Leeds, then $h(\text{Keswick})$ tells me approximately how much longer I have to cycle.

Examples

- The function $h\colon V \to \mathbb{Q}_{\geq 0}$ with $h(v) = 0$ for all nodes $v \in V$
- If our nodes are in the Euclidean (i. e. 2-dimensional) plain, then the straight-line distance, given by
$$h(v) = \sqrt{(v_x - t_x)^2 + (v_y - t_y)^2}$$
for all $v \in V$ is a heuristic function.

Application

- A heuristic for the distance from any city to the target could be the distance on a map measured using a ruler.

Admissible heuristic: definition

A heuristic function is **admissible**, if it never overestimates the distance to the goal, i. e. all nodes v satisfy: $h(v) \leq$ length of a shortest v-t-path.

Examples

- The function $h\colon V \to \mathbb{Q}_{\geq 0}$ with $h(v) = 0$ for all nodes $v \in V$
- If our nodes are in the Euclidean plain, then the straight-line distance, given by

$$h(v) = \sqrt{(v_x - t_x)^2 + (v_y - t_y)^2}$$

  is admissible (proof idea: next slide).

Monotone heuristics

- Let $u, v \in V$ with $(u, v) \in E$. A heuristic h is called **monotone** (or **consistent**), if the following is satisfied:

$$h(u) \leq w(u, v) + h(v).$$

Monotone heuristics

- Let $u, v \in V$ with $(u, v) \in E$. A heuristic h is called **monotone** (or **consistent**), if the following is satisfied:

$$h(u) \leq w(u, v) + h(v).$$

- Note that this resembles the triangle-inequality.

Monotone heuristics

- Let $u, v \in V$ with $(u, v) \in E$. A heuristic h is called **monotone** (or **consistent**), if the following is satisfied:

$$h(u) \leq w(u, v) + h(v).$$

- Note that this resembles the triangle-inequality.

Example: The straight-line heuristic is monotone. (Proof: exercise.)

Monotone heuristics

- Let $u, v \in V$ with $(u, v) \in E$. A heuristic h is called **monotone** (or **consistent**), if the following is satisfied:

$$h(u) \leq w(u, v) + h(v).$$

- Note that this resembles the triangle-inequality.

Example: The straight-line heuristic is monotone. (Proof: exercise.)

Remark:

- Every heuristic that is monotone is also admissible. (Proof: exercise.)
- Admissible heuristics can over-estimate the distance between two nodes, whereas monotone heuristics cannot. This means that the first path discovered to a node v will also be the shortest – hence unnecessary 'detours' are avoided.

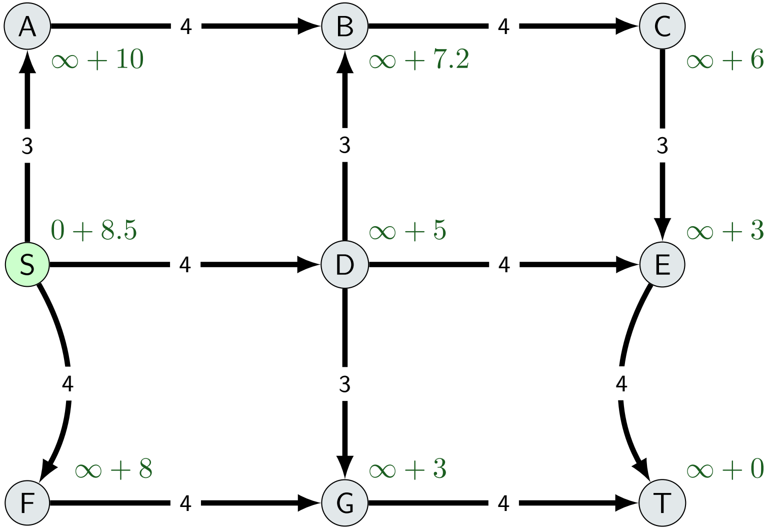
A^* search algorithm: pseudo-code

input : graph $D = (V, E)$, $w : E \to \mathbb{Q}_{\geq 0}$, **monotone $h : V \to \mathbb{Q}_{\geq 0}$** , start vertex s, and **target vertex t**.

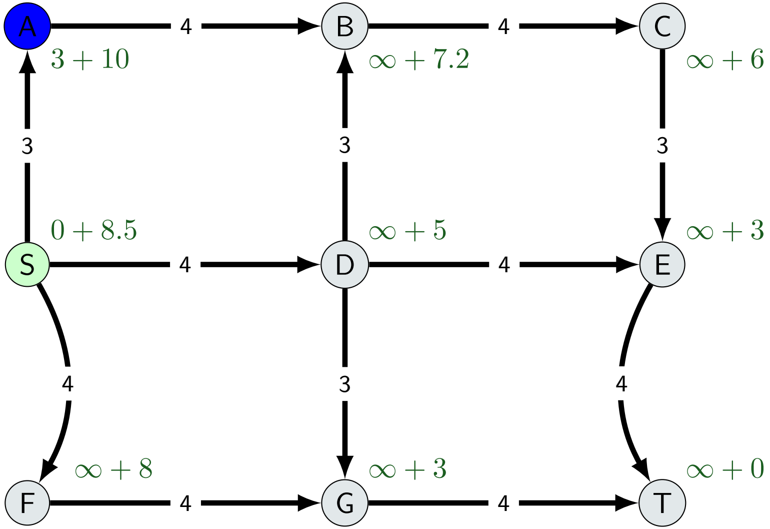
output : $l[t]$, the distance from s to t, shortest s-t-path in (V, F) if exists

```
procedure AStar $(D, w, s)$
    for $v \in V \setminus \{s\}$ do visited$[v]$ $\leftarrow$false; $l[v] \leftarrow \infty$ ;
    $L \leftarrow V \setminus \{s\}$; $F \leftarrow \varnothing$; visited$[s]$ $\leftarrow$true; $l[s] \leftarrow 0$;
    for $v \in N^+(s)$ do $l[v] \leftarrow w(s, v)$; $p[v] \leftarrow s$;
    while $L \neq \varnothing$ do
        choose $u \in L$ with smallest $l[u] + h(u)$;
        $L \leftarrow L \setminus \{u\}$; $F \leftarrow F \cup \{(p[u], u)\}$; visited$[u]$ $\leftarrow$true;
        if $u = t$ then break;
        for $v \in N^+(u)$ do
            if !visited$[v]$ and $l[v] > l[u] + w(u, v)$ then
                $l[v] \leftarrow l[u] + w(u, v)$; $p[v] \leftarrow u$
```

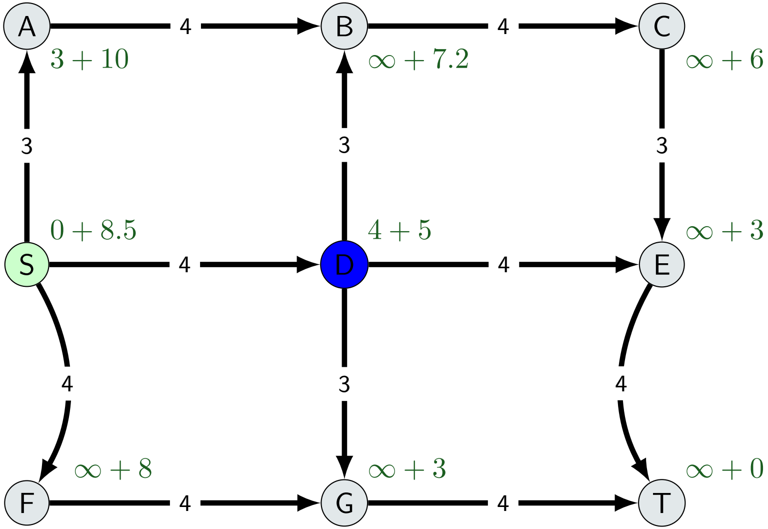
A^* on grid: example



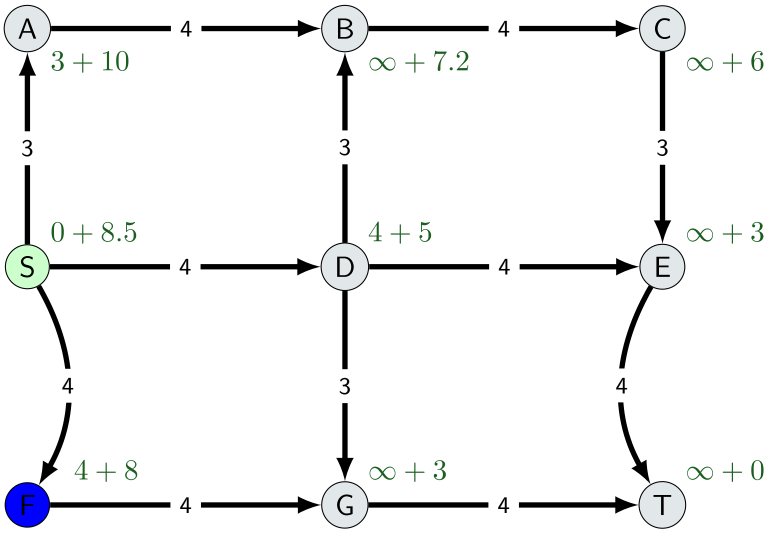
A^* on grid: example



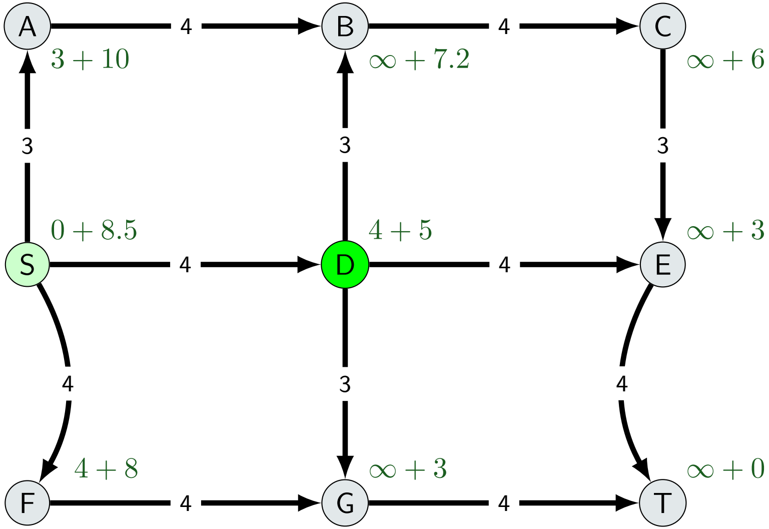
A^* on grid: example



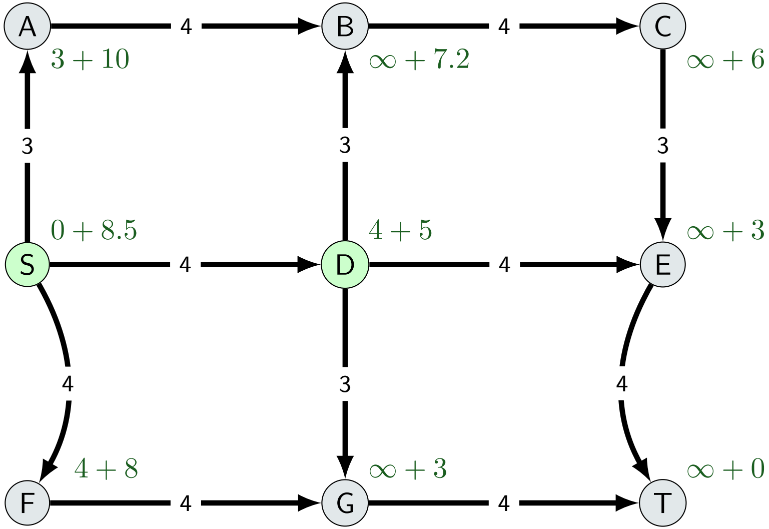
A^* on grid: example



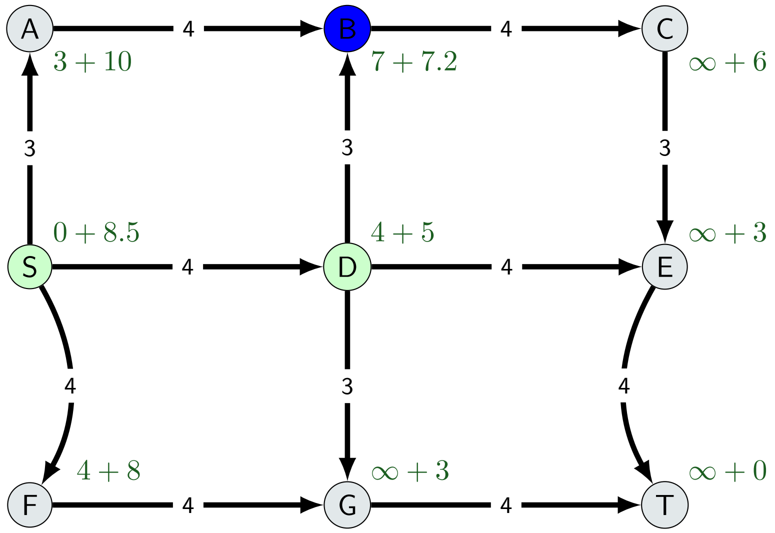
A^* on grid: example



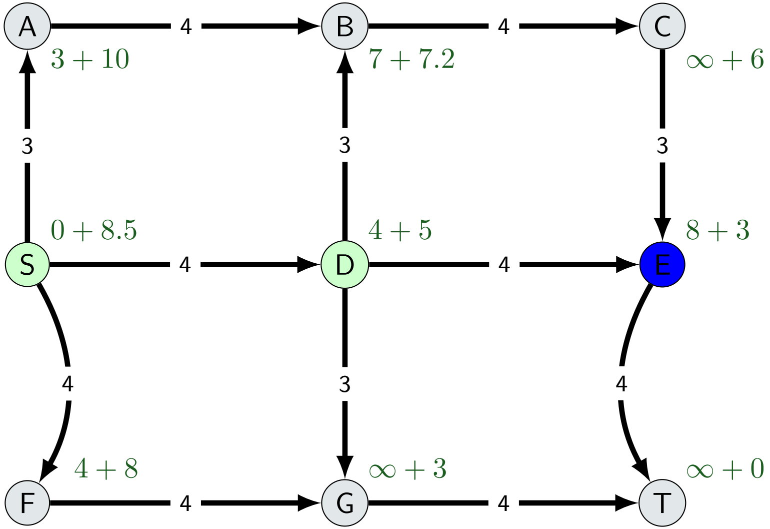
A^* on grid: example



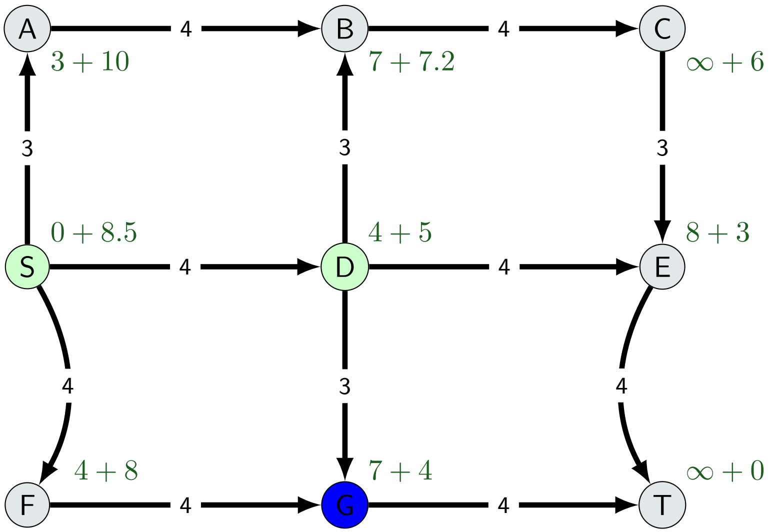
A^* on grid: example



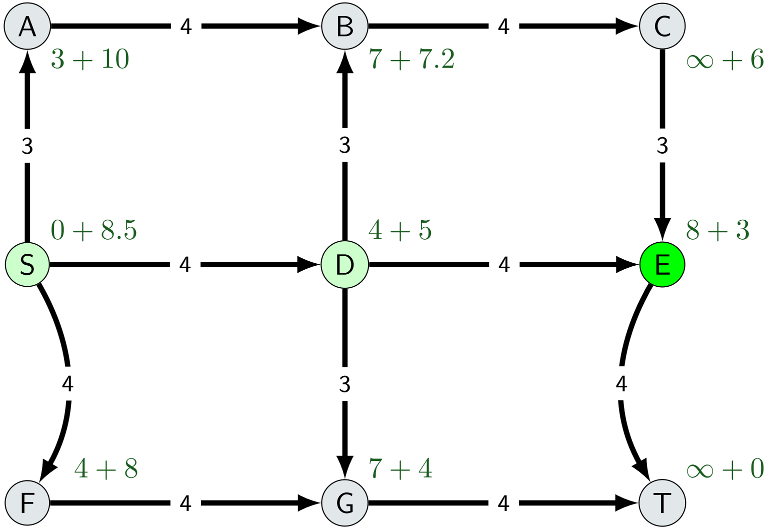
A^* on grid: example



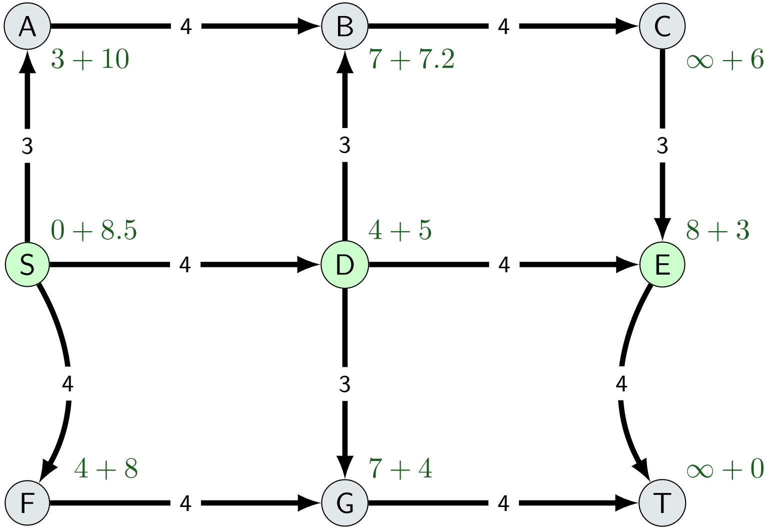
A^* on grid: example



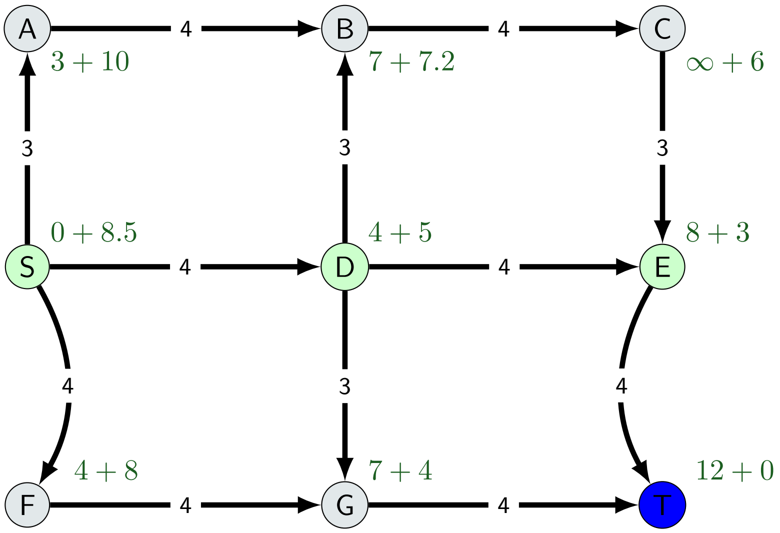
A^* on grid: example



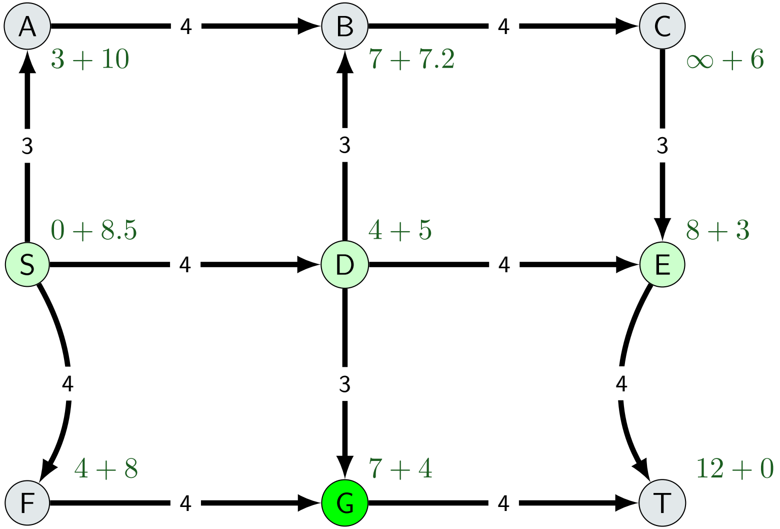
A^* on grid: example



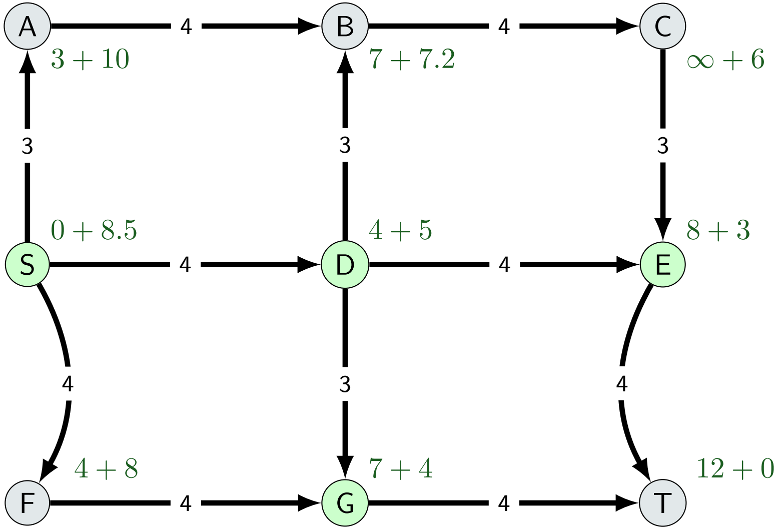
A^* on grid: example



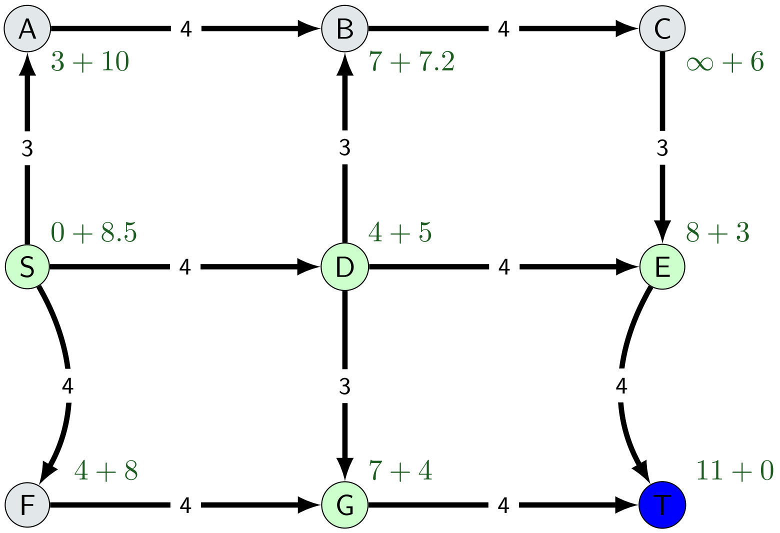
A^* on grid: example



A^* on grid: example



A^* **on grid: example**



Correctness of A^*

Theorem: If h is monotone, then A^* always finds an optimal s-t-path, i. e. a path with minimal length, if an s-t-path exists.

Correctness of A^*

Theorem: If h is monotone, then A^* always finds an optimal s-t-path, i. e. a path with minimal length, if an s-t-path exists.

Proof sketch:

- Idea: Show that A^* is the same as Dijkstra (with suitable edge weights w').

Correctness of A^*

Theorem: If h is monotone, then A^* always finds an optimal s-t-path, i. e. a path with minimal length, if an s-t-path exists.

Proof sketch:

- Idea: Show that A^* is the same as Dijkstra (with suitable edge weights w').
- Define $w'\colon E \to \mathbb{Q}_{\geq 0}$ as $w'(u,v) = w(u,v) + h(v) - h(u)$.

Correctness of A^*

Theorem: If h is monotone, then A^* always finds an optimal s-t-path, i. e. a path with minimal length, if an s-t-path exists.

Proof sketch:

- Idea: Show that A^* is the same as Dijkstra (with suitable edge weights w').
- Define $w' \colon E \to \mathbb{Q}_{\geq 0}$ as $w'(u, v) = w(u, v) + h(v) - h(u)$.
- Then $w'(u, v) \geq 0$ for all $(u, v) \in E$ (h is monotone). Hence suitable for Dijkstra.

Correctness of A^*

Theorem: If h is monotone, then A^* always finds an optimal s-t-path, i. e. a path with minimal length, if an s-t-path exists.

Proof sketch:

- Idea: Show that A^* is the same as Dijkstra (with suitable edge weights w').
- Define $w' \colon E \to \mathbb{Q}_{\geq 0}$ as $w'(u, v) = w(u, v) + h(v) - h(u)$.
- Then $w'(u, v) \geq 0$ for all $(u, v) \in E$ (h is monotone). Hence suitable for Dijkstra.
- The new length of any u-v-path P is $\sum_{e \in P} w'(e) = \sum_{e \in P} w(e) + h(v) - h(u)$.
  If P is an s-t-path, then $\sum_{e \in P} w'(e) = \sum_{e \in P} w(e) - h(s)$ (because $h(t) = 0$).

Correctness of A^*

Theorem: If h is monotone, then A^* always finds an optimal s-t-path, i. e. a path with minimal length, if an s-t-path exists.

Proof sketch:

- Idea: Show that A^* is the same as Dijkstra (with suitable edge weights w').
- Define $w' \colon E \to \mathbb{Q}_{\geq 0}$ as $w'(u, v) = w(u, v) + h(v) - h(u)$.
- Then $w'(u, v) \geq 0$ for all $(u, v) \in E$ (h is monotone). Hence suitable for Dijkstra.
- The new length of any u-v-path P is $\sum_{e \in P} w'(e) = \sum_{e \in P} w(e) + h(v) - h(u)$.
  If P is an s-t-path, then $\sum_{e \in P} w'(e) = \sum_{e \in P} w(e) - h(s)$ (because $h(t) = 0$).
- W. l. o. g. $h(s) = 0$.

Correctness of A^*

Theorem: If h is monotone, then A^* always finds an optimal s-t-path, i. e. a path with minimal length, if an s-t-path exists.

Proof sketch:

- Idea: Show that A^* is the same as Dijkstra (with suitable edge weights w').
- Define $w' \colon E \to \mathbb{Q}_{\geq 0}$ as $w'(u,v) = w(u,v) + h(v) - h(u)$.
- Then $w'(u,v) \geq 0$ for all $(u,v) \in E$ (h is monotone). Hence suitable for Dijkstra.
- The new length of any u-v-path P is $\sum_{e \in P} w'(e) = \sum_{e \in P} w(e) + h(v) - h(u)$.
  If P is an s-t-path, then $\sum_{e \in P} w'(e) = \sum_{e \in P} w(e) - h(s)$ (because $h(t) = 0$).
- W. l. o. g. $h(s) = 0$.
- Observaton: P is a shortest s-t-path w.r.t. w iff P is a shortest s-t-path w.r.t. w'.

Correctness of A^*

Theorem: If h is monotone, then A^* always finds an optimal s-t-path, i. e. a path with minimal length, if an s-t-path exists.

Proof sketch:

- Idea: Show that A^* is the same as Dijkstra (with suitable edge weights w').
- Define $w' \colon E \to \mathbb{Q}_{\geq 0}$ as $w'(u,v) = w(u,v) + h(v) - h(u)$.
- Then $w'(u,v) \geq 0$ for all $(u,v) \in E$ (h is monotone). Hence suitable for Dijkstra.
- The new length of any u-v-path P is $\sum_{e \in P} w'(e) = \sum_{e \in P} w(e) + h(v) - h(u)$.
  If P is an s-t-path, then $\sum_{e \in P} w'(e) = \sum_{e \in P} w(e) - h(s)$ (because $h(t) = 0$).
- W. l. o. g. $h(s) = 0$.
- Observaton: P is a shortest s-t-path w.r.t. w iff P is a shortest s-t-path w.r.t. w'.
- Finally: verify that running A^* on D with weight function w and heuristic h is the same as running Dijkstra's algorithm on D with weight function w'.

$\square$

A^* search algorithm: remarks

Remarks

- The worst-case running time for A^* is the same as for Dijkstra's algorithm.
- A^* generalises Dijkstra's algorithm for finding a shortest s-t-path:
  Let $h(v) = 0$ for every node v to obtain Dijkstra's algorithm.
- In practice, A^* can be **much faster** than Dijkstra's algorithm.
  Examples: cities in the Euclidean plain, together with the straight-line distance; search in a grid.

Original A^* algorithm: remark

- The A^* search algorithm was first proposed in:

  A formal basis for the heuristic determination of Minimum Cost Paths, by Peter Hart, Nils Nilsson, and Bertram Raphael, IEEE Transactions on systems science and cybernetics, 1968.

- The algorithm works for admissible (not necessarily monotone) heuristics.

- The correctness proof is more involved.

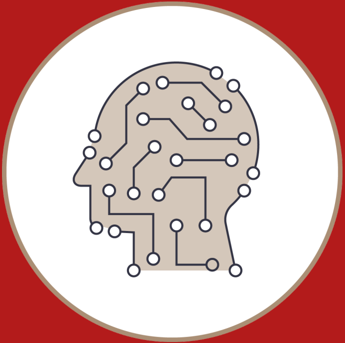
- The running time depends on the heuristic.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Divide and conquer: introduction

Isolde Adler and Sebastian Ordyniak



Algorithms: paradigms

Greedy: Build a solution incrementally by choosing the next solution component using a local and not forward-looking criteria.

Divide and conquer: Break up the problem into subproblems. Solve subproblems independently and combine solutions for the subproblems to a solution for the whole problem.

Divide and conquer: introduction

The **divide and conquer** technique leads to recursive algorithms. This paradigm involves three steps at each level of the recursion:

Divide: the current instance into a number of subinstances of the same problem, but smaller in size.

Conquer: the subinstances by solving them recursively. If an instance is small enough solve it directly.

Combine: the solutions to the sub-instances into a solution for the original instance.

Divide and conquer algorithms are recursive.

Examples: binary search, merge sort, quick sort

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Divide and conquer: sorting (introduction)

Isolde Adler and Sebastian Ordyniak

Sorting: Given n elements, rearrange in ascending order.

Applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

obvious applications

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- . . .

non-obvious applications

Sorting: basics

Simple sorting algorithms:

- Bubblesort
- Insertionsort
- Selectionsort

Run-time: Worst-case and average-case $\Theta(n^2)$.

Question: Can one sort faster?

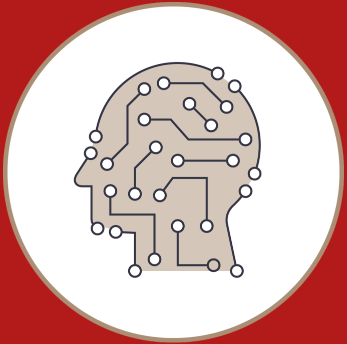
Answer: Yes. Mergesort can sort in $\Theta(n \log n)$ time (and this is basically optimal).

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

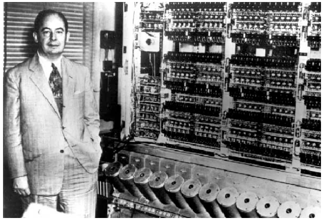
Divide and conquer: sorting (mergesort)

Isolde Adler and Sebastian Ordyniak



Mergesort

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.


Jon von Neumann (1945)

A L G O R I T H M S

A L G O R   I T H M S   divide $\mathcal{O}(1)$

A G L O R   H I M S T   sort $2T(n/2)$

A G H I L M O R S T   merge $\mathcal{O}(n)$

Mergesort

```
input   : array $A$, integers $l$ and $r$.
effect  : elements in $A$ between $l$ and $r$ are sorted.

procedure mergesort$(A, l, r)$
    if $l < r$ then
        m $\leftarrow \lfloor (l + r)/2 \rfloor$;
        mergesort$(A, l, m)$;
        mergesort$(A, m + 1, r)$;
        merge$(A, l, m, r)$;
```

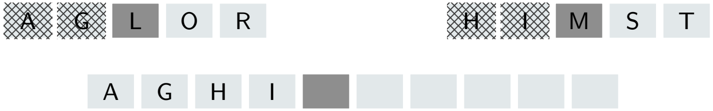
Call: `mergesort`$(A, 0, n - 1)$ for an array A with n elements.

Merging

Idea: Merge two sorted lists into a sorted combined list.

How to merge efficiently?

- Use a temporary array.
- Iterate through both lists from the beginning.
- Merge the elements of the two lists in a zipper style, always taking the smaller element from the two lists.
- Linear runtime.

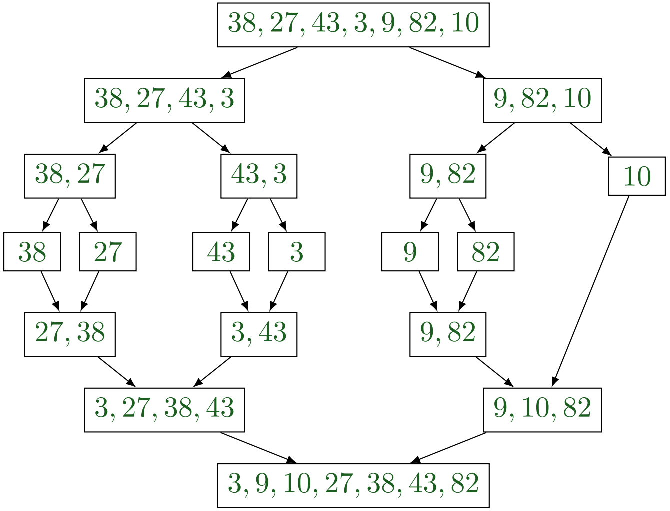


Merging: pseudocode

input : array A, integers l, m, and r (elements between l and m as well as between m and r must already be sorted).
effect : elements in A between l and r are sorted.

```
procedure merge$(A, l, m, r)$
    $i \leftarrow l$; $j \leftarrow m + 1$; $k \leftarrow l$;
    while $i \leq m$ and $j \leq r$ do
        if A[$i$] $\leq$ A[$j$] then
            B[$k$] $\leftarrow$ A[$i$]; $i \leftarrow i + 1$;
        else
            B[$k$] $\leftarrow$ A[$j$]; $j \leftarrow j + 1$;
        $k \leftarrow k + 1$;
    if $i > m$ then
        for $h \leftarrow j$ to $r$ do
            B[$k$] $\leftarrow$ A[$h$]; $k \leftarrow k + 1$;
    else
        for $h \leftarrow i$ to $m$ do
            B[$k$] $\leftarrow$ A[$h$]; $k \leftarrow k + 1$;
    for $h \leftarrow l$ to $r$ do
        A[$h$] $\leftarrow$ B[$h$];
```

Mergesort: example



A useful recurrence relation

Definition: $T(n) =$ number of comparisons to mergesort an input of size n. Note that $T(n)$ is also proportional to the runtime of mergesort.

Mergesort recurrence:

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T\left(\lceil n/2 \rceil\right)}_{\text{solve left half}} + \underbrace{T\left(\lfloor n/2 \rfloor\right)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution: $T(n) = \mathcal{O}(n \log_2 n)$.

Remark: Since the runtime of mergesort is dominated by the number $T(n)$ of comparisons, $T(n)$ is also proportional to the runtime of mergesort.

A useful recurrence relation

Proof: We provide several ways to show that $T(n) = \mathcal{O}(n \log_2 n)$.

Assumptions:

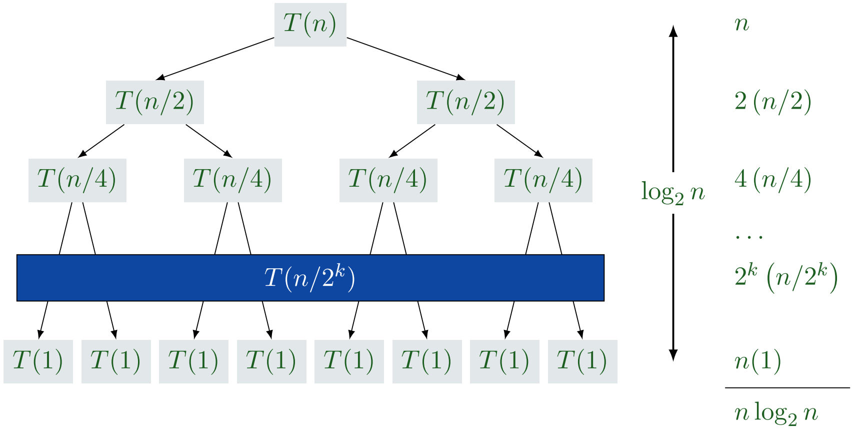
- We will start by assuming that n is a power of 2.
- For an arbitary n' with $\frac{n}{2} < n' < n$ (where n is a power of 2) it then holds that:

$$T(n') = \mathcal{O}\left(n' \log n'\right)$$

since $\mathcal{O}(\frac{n}{2} \log \frac{n}{2}) = \mathcal{O}(n \log n)$.

Proof by recursion tree

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Proof by telescoping

Claim: If $T(n)$ satisfies this recurrence, then $T(n) = n \log_2 n$.

- *assumes n is a power of 2*

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Proof: For $n > 1$:

$$\begin{aligned} \frac{T(n)}{n} &= \frac{2T(n/2)}{n} && + 1 \\ &= \frac{T(n/2)}{n/2} && + 1 \\ &= \frac{T(n/4)}{n/4} && + 1 + 1 \\ &= \frac{T(n/n)}{n/n} && + \underbrace{1 + \cdots + 1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

Proof by induction

Claim: If $T(n)$ satisfies this recurrence, then $T(n) = n \log_2 n$.

🟩 *assumes n is a power of 2*

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T\left(n/2\right)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Proof: (by induction on n)

- Base case: $n = 1$.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: Show that $T(2n) = 2n \log_2(2n)$.

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n(\log_2(2n) - 1) + 2n \\ &= 2n \log_2(2n) \end{aligned}$$

Analysis of mergesort recurrence

Claim: If $T(n)$ satisfies the following recurrence, then $T(n) \leq n\lceil \log_2 n \rceil$.

$$T(n) \leq \begin{cases} 0 \\ \underbrace{T\left(\lceil n/2 \rceil\right)}_{\text{solve left half}} + \underbrace{T\left(\lfloor n/2 \rfloor\right)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} \end{cases}$$

Proof: (by induction on n)

- Base case: $n = 1$.
- Define $n_1 = \lfloor n/2 \rfloor$, $n_2 = \lceil n/2 \rceil$.
- Induction step: Assume true for $1, 2, \ldots, n-1$.

$$\begin{aligned} T(n) &\leq T(n_1) + T(n_2) + n \\ &\leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\ &= n \lceil \log_2 n_2 \rceil + n \\ &\leq n \left(\lceil \log_2 n \rceil - 1 \right) + n \\ &= n \lceil \log_2 n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\leq \lceil 2^{\lceil \log_2 n \rceil} / 2 \rceil \\ &= 2^{\lceil \log_2 n \rceil} / 2 \\ &\Rightarrow \log_2 n_2 \leq \lceil \log_2 n \rceil - 1 \end{aligned}$$

Analysis of the mergesort recurrence

Since the merging of two sorted lists requires also **at least** $\lfloor \frac{n}{2} \rfloor$, we obtain that:

$$T_{\text{best}}(n) = T_{\text{worst}}(n) = T_{\text{avg}}(n) = \Theta(n \log n)$$

That is, the runtime for mergesort is the same for best-case, average-case, and worst-case.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Divide and conquer: sorting (quicksort)

Isolde Adler and Sebastian Ordyniak

Quicksort

Also uses divide and conquer, however, slightly different.

Divide: Choose a **pivot element** x from the sequence A, e. g. the first/last element.
Split A without x into two subsequences A_L and A_R such that:

- A_L contains all elements that are at most x ($\leq x$).
- A_R contains all elements that are larger than x ($> x$).

Conquer:

- Recurse for A_L.
- Recurse for A_R.

Combine: Obtain the sorted list as A_L, x, A_R.

Quicksort: pseudocode

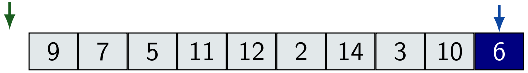
```
input    : array $A$ and integers $l$ and $r$.
effect   : elements in $A$ between $l$ and $r$ are sorted.

procedure quicksort($A, l, r$)
    if $l < r$ then
        /* choose last element as the pivot element; other strategies
           are possible                                                */
        $p \leftarrow A[r]$;
        $s \leftarrow$partition($A, l, r, r$);
        // $p$ is at position $s$, all elements before/after $s$ are $\leq p$/$\geq p$.
        quicksort($A, l, s - 1$);
        quicksort($A, s + 1, r$);
```

Call: quicksort$(A, 0, n - 1)$ for an array A with n elements.

Partition: Idea

0) Start from both sides, using two indices i and j (initially $i = l - 1$ and $j = r$).
1) Increase i until reaching an element larger than the pivot.
2) Decrease j until reaching an element smaller than the pivot.
3) Swap $A[i]$ and $A[j]$.
4) Repeat steps 1)–3) until $j \leq i$.



Partition: Idea

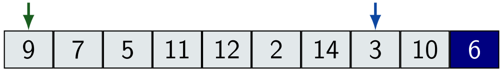
0) Start from both sides, using two indices i and j (initially $i = l - 1$ and $j = r$).

1) Increase i until reaching an element larger than the pivot.

2) Decrease j until reaching an element smaller than the pivot.

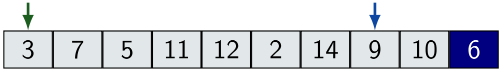
3) Swap $A[i]$ and $A[j]$.

4) Repeat steps 1)–3) until $j \leq i$.



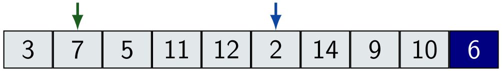
Partition: Idea

0) Start from both sides, using two indices i and j (initially $i = l - 1$ and $j = r$).
1) Increase i until reaching an element larger than the pivot.
2) Decrease j until reaching an element smaller than the pivot.
3) Swap $A[i]$ and $A[j]$.
4) Repeat steps 1)–3) until $j \leq i$.



Partition: Idea

0) Start from both sides, using two indices i and j (initially $i = l - 1$ and $j = r$).
1) Increase i until reaching an element larger than the pivot.
2) Decrease j until reaching an element smaller than the pivot.
3) Swap $A[i]$ and $A[j]$.
4) Repeat steps 1)–3) until $j \leq i$.



Partition: Idea

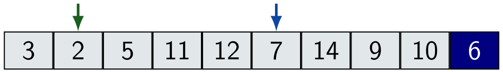
0) Start from both sides, using two indices i and j (initially $i = l - 1$ and $j = r$).

1) Increase i until reaching an element larger than the pivot.

2) Decrease j until reaching an element smaller than the pivot.

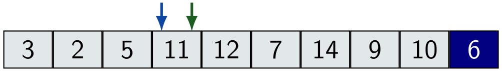
3) Swap $A[i]$ and $A[j]$.

4) Repeat steps 1)–3) until $j \leq i$.



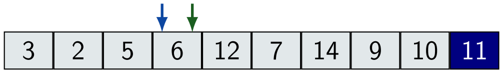
Partition: Idea

0) Start from both sides, using two indices i and j (initially $i = l - 1$ and $j = r$).
1) Increase i until reaching an element larger than the pivot.
2) Decrease j until reaching an element smaller than the pivot.
3) Swap $A[i]$ and $A[j]$.
4) Repeat steps 1)–3) until $j \leq i$.



Partition: Idea

0) Start from both sides, using two indices i and j (initially $i = l - 1$ and $j = r$).
1) Increase i until reaching an element larger than the pivot.
2) Decrease j until reaching an element smaller than the pivot.
3) Swap $A[i]$ and $A[j]$.
4) Repeat steps 1)–3) until $j \leq i$.

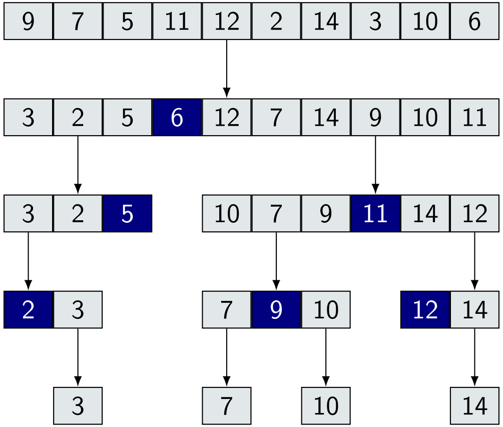


Partition: pseudocode

input : array A, integers l, r, and the index s of the pivot element in A.
output : index i of pivot element after the reorganisation.
effect : all elements in A before i are $\leq A[s]$ and all elements after i are $> A[s]$.

```
function partition($(A, l, r, s)$)
    $i \leftarrow l - 1$; $j \leftarrow r$;
    swap $A[s]$ and $A[r]$;
    repeat
        repeat $i \leftarrow i + 1$ until $j \leq i$ or $A[i] > A[r]$;
        ;
        repeat $j \leftarrow j - 1$ until $j \leq i$ or $A[j] \leq A[r]$;
        ;
        if $i < j$ then
            swap $A[i]$ and $A[j]$;
    until $i \geq j$;
    swap $A[i]$ and $A[r]$;
    return $i$;
```

Quicksort: example



Quicksort: analysis

Recurrence: (Depends strongly on the choice of the pivot element)

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(n-r-1)}_{\text{solve left half}} + \underbrace{T(r)}_{\text{solve right half}} + \underbrace{n}_{\text{partitioning}} & \text{otherwise} \end{cases}$$

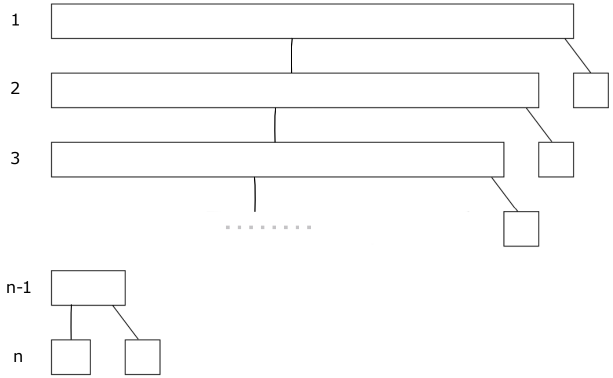
$r \approx$ number r of elements that are larger than the pivot

Best-case:

- Both subsequences almost always have approximately the same size.
- Then, recurrence and therefore the run-time is the same as Mergesort, i.e., $\Theta(n \log n)$.

Quicksort: analysis

Worst-case: Every (sub-)sequence is split at the smallest/largest element.



Then, the runtime is $\Theta(n^2)$.

Quicksort: analysis

Worst-case: possible scenario:

- Already sorted sequence and the algorithm chooses the last element as pivot.
- Then, all other elements are smaller than the pivot and therefore the left subsequence contains all elements but the pivot and the right subsequence is empty.
- Therefore, the length of the subsequence at the next recursive step merely decreases by 1!
- $\Theta(n)$ recursive calls.
- Run-time in $\Theta(n^2)$.
- The additional memory required for the $\Theta(n)$ recursive calls is $\Theta(n)$.

Quicksort: analysis

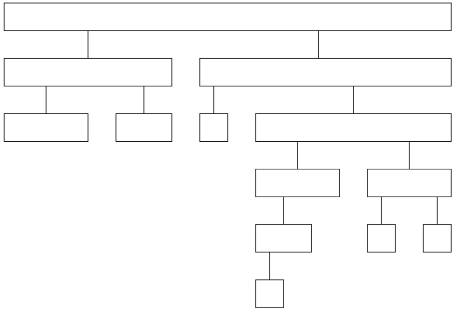
How to avoid the worst-case in practice:

- Choose a random pivot.
    - **Randomised quicksort**.
    - The sorted list is no longer the worst-case scenario.
- Choose three (or a constant number) of elements and use the median of these elements as pivot.

Quicksort: analysis

Average-case: Using a rather complicated proof one can show that the runtime of quicksort is $\Theta(n \log n)$ on average.

Example for average-case: Every (sub-)sequence is split close to the middle.



Memory usage

Memory usage: A measure for how the memory usage grows as a function of the input size.

- Quicksort: Worst-case in $\Theta(n)$, best/average-case in $\Theta(\log n)$.
- Mergesort: Best/average/worst-case in $\Theta(n)$.

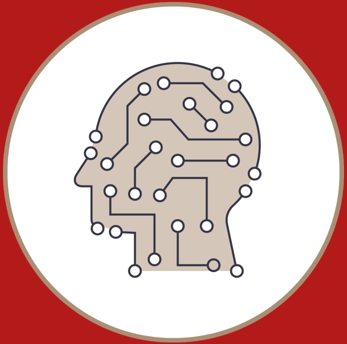
Practice: This is the main reason quicksort is preferred to mergesort for most practical applications.

UNIVERSITY OF LEEDS

OCOM5102 – Algorithms

Divide and conquer: sorting (comparison)

Isolde Adler and Sebastian Ordyniak



Comparison of sorting methods

Run-time comparisons

Sorting method	**Best-case**	**Average-case**	**Worst-case**
insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$

Additional memory usage

Sorting method	**Best-case**	**Average-case**	**Worst-case**
insertionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
selectionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
mergesort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
quicksort	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$

Sorting methods by application

Quicksort:

- Often the preferred method for most use cases.

Mergesort:

- Mergesort is mainly used for sorting lists.
- Also used for sorting elements on external memory:
  - Here one uses an iterative (instead of recursive) version of mergesort (e.g., bottom-up mergesort) that only requires $\mathcal{O}(\log n)$ passes through a file.

UNIVERSITY OF LEEDS