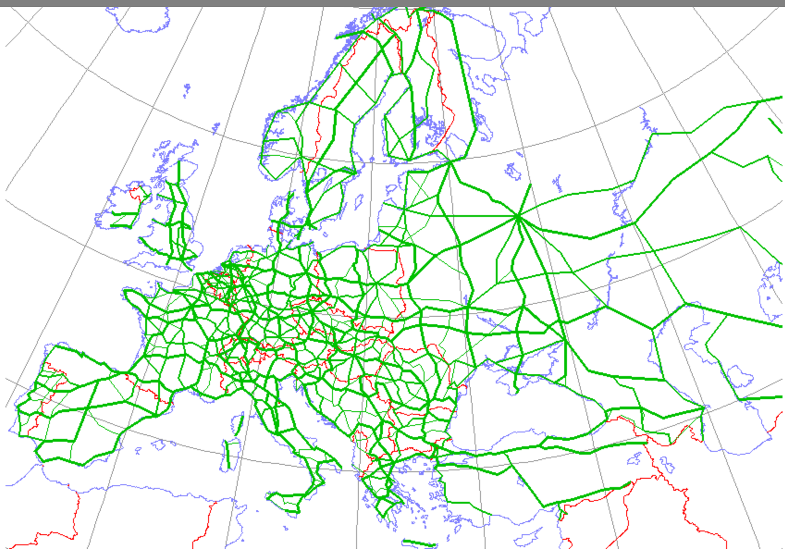
Fundamental Graph Algorithms

KSETA · March 9, 2020
Demian Hespe, Tobias Heuer and Sebastian Lamm

INSTITUTE OF THEORETICAL INFORMATICS · ALGORITHMICS GROUP



Outline

- Foundations — **1. Session**
  - Complexity Theory
  - Graph Notation/Properties
  - Graph Representation
  - Graph Exploration
- The Good, Bad & Ugly — **2. Session**
- Network Analysis — **3. Session**
- Case Studies in Physics
- Network Analysis Tutorial — **4. Session**

The Good, Bad & Ugly

The Good

- Shortest Paths
- Minimum Spanning Trees
- Maximum Flows
- Maximum Matchings

The Bad & Ugly

- Coloring
- Traveling Salesman
- Independent Sets
- (Hyper-)graph Partitioning

Institute of Theoretical Informatics
Algorithmics Group

The Good, Bad & Ugly

The Good

- Shortest Paths
- Minimum Spanning Trees
- Maximum Flows
- Maximum Matchings

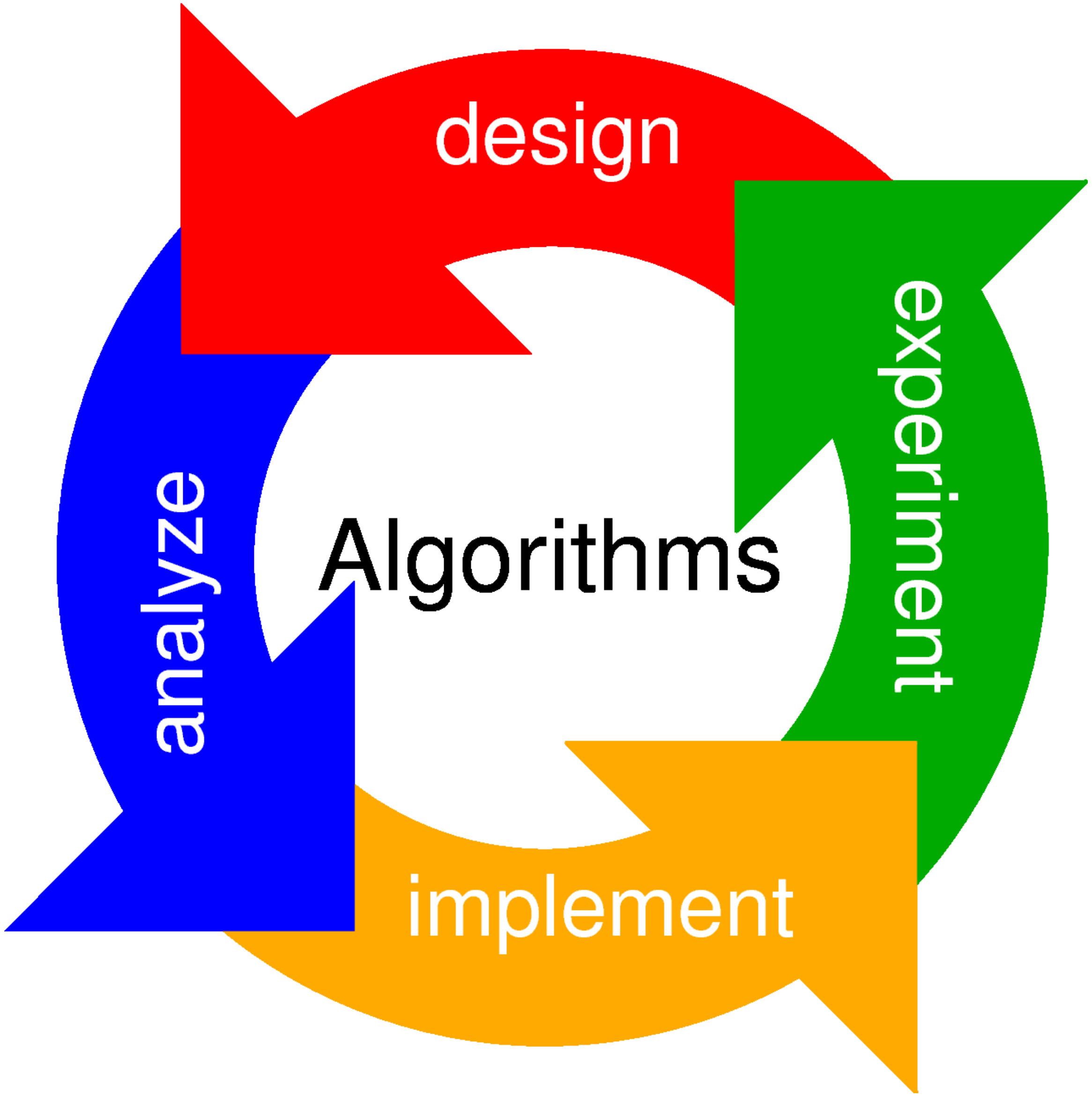
The Bad & Ugly

- Coloring
- Traveling Salesman
- Independent Sets
- (Hyper-)graph Partitioning

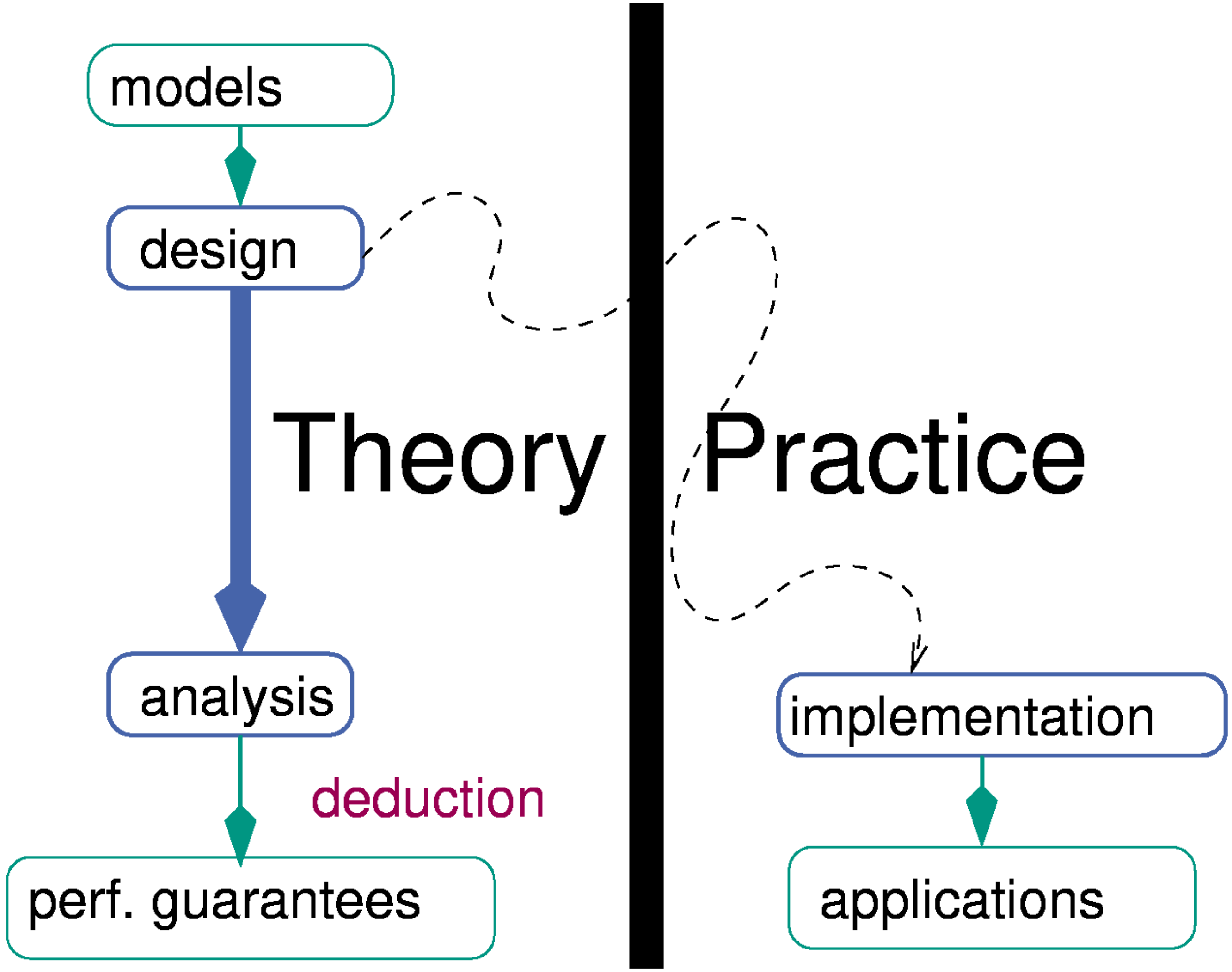
slides available at:
http://algo2.iti.kit.edu/documents/graph_theory.pdf

Algorithm Engineering

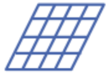
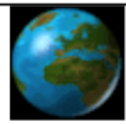
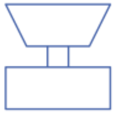
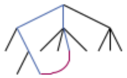
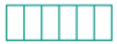
Algorithm Engineering



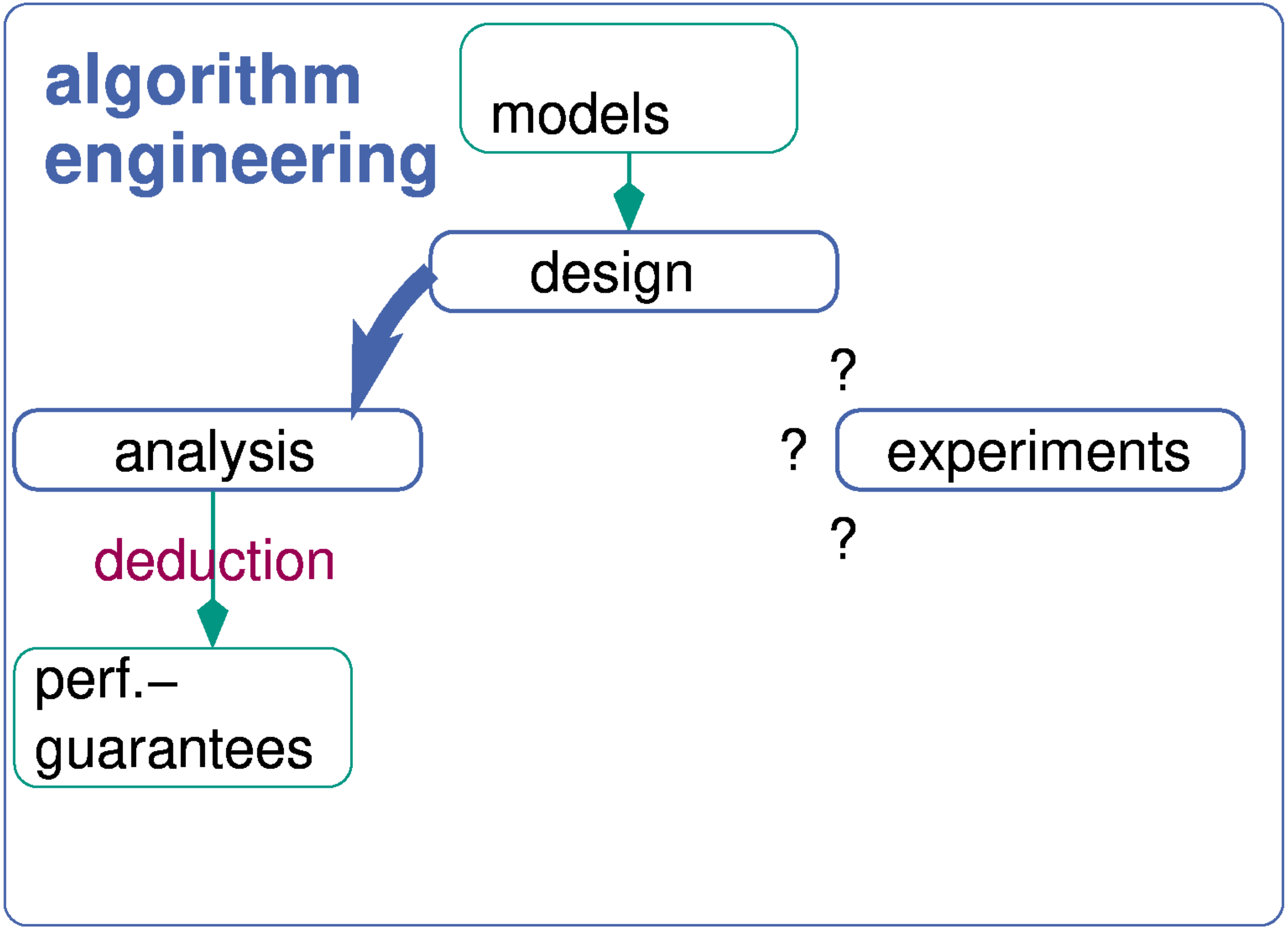
(Caricatured) Traditional View: Algorithm Theory



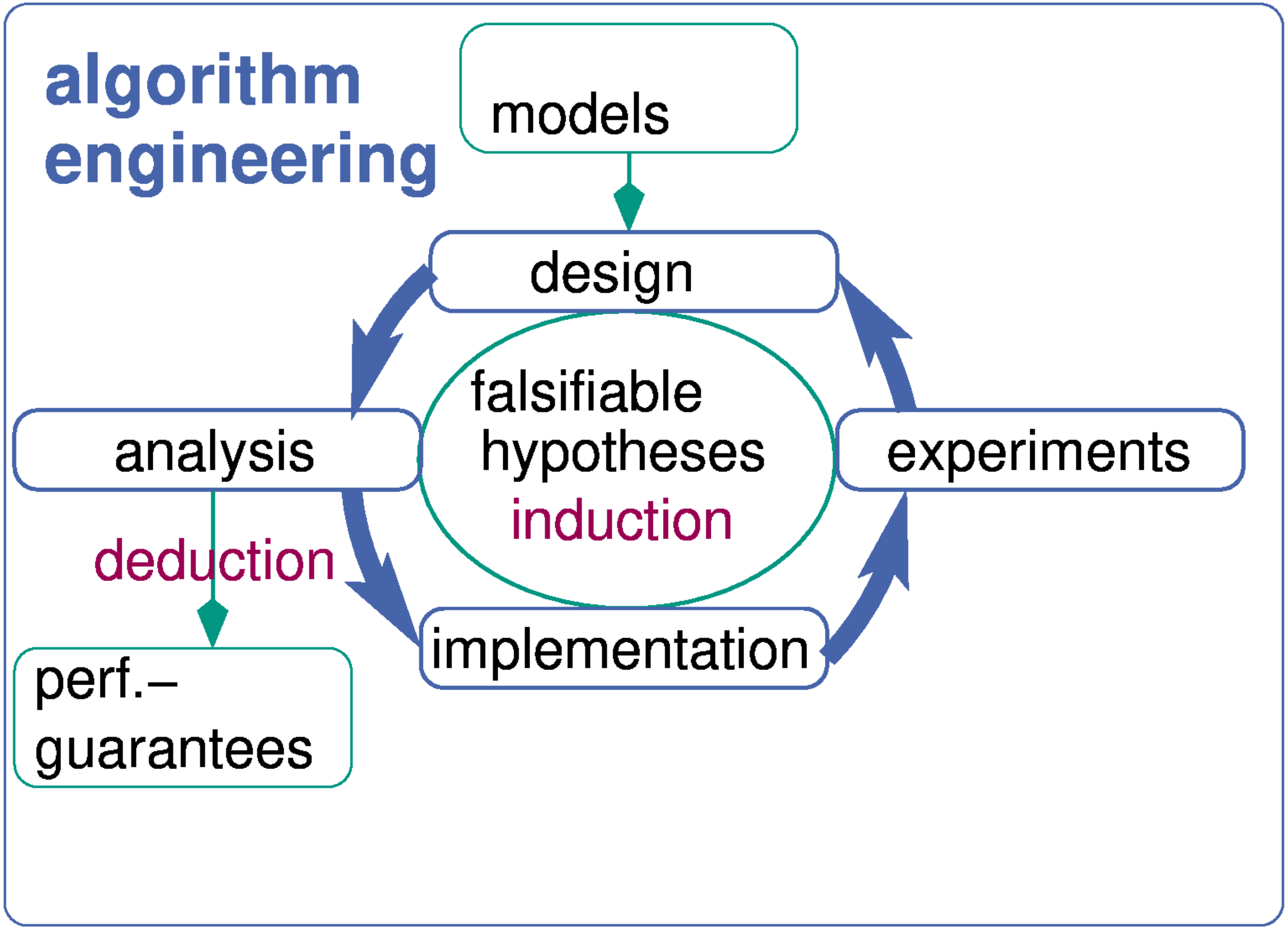
Gaps Between Theory & Practice

Theory		$\longleftrightarrow$		Practice
simple		**appl. model**		complex
simple		**machine model**		real
complex		**algorithms**	$\boxed{\texttt{FOR}}$	simple
advanced		**data structures**		arrays, . . .
worst case	$\boxed{\text{max}}$	**complexity measure**		inputs
asympt.	$\boxed{O(\cdot)}$	**efficiency**	$\boxed{42\%}$	constant factors

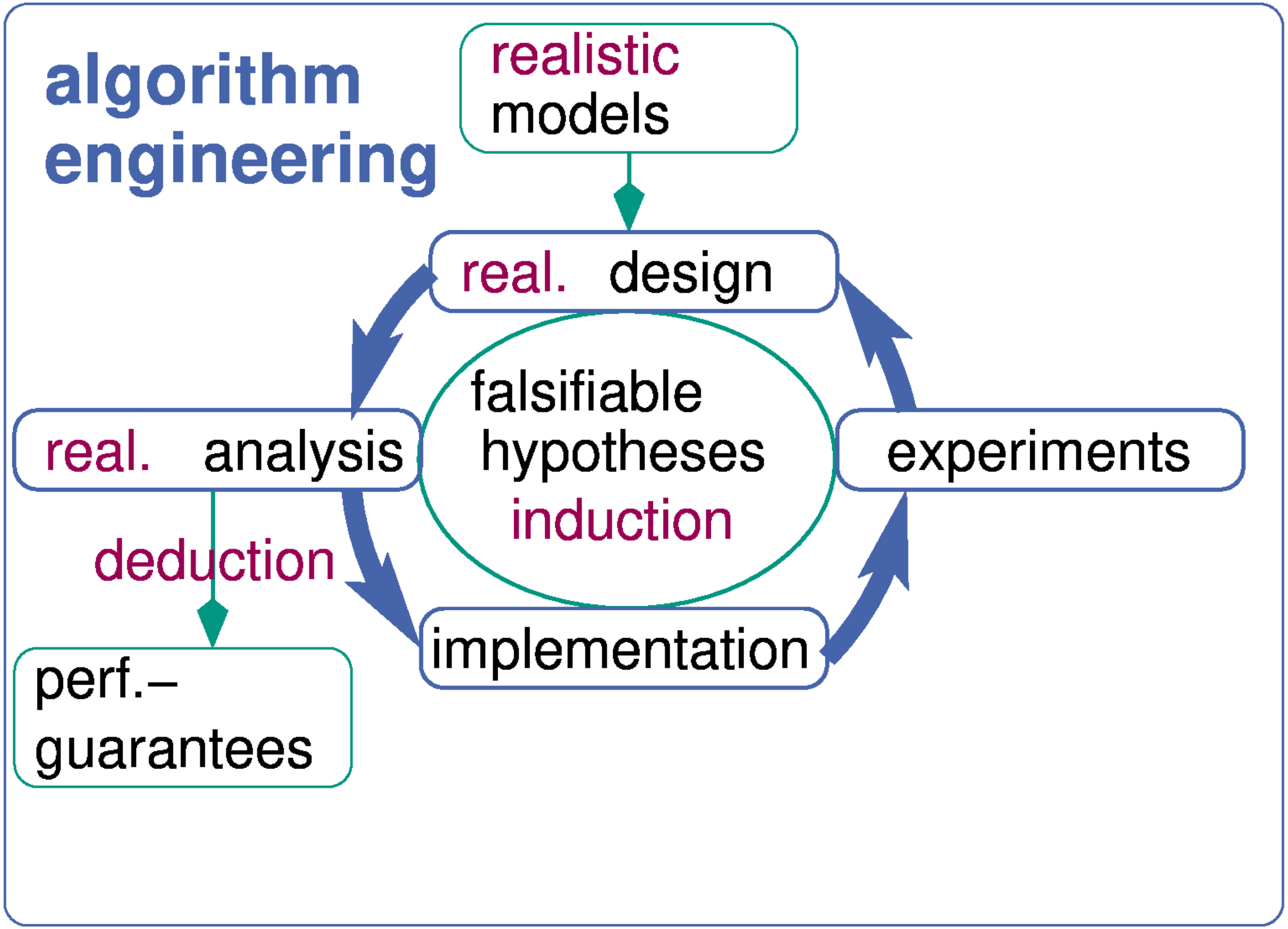
Algorithmics as Algorithm Engineering



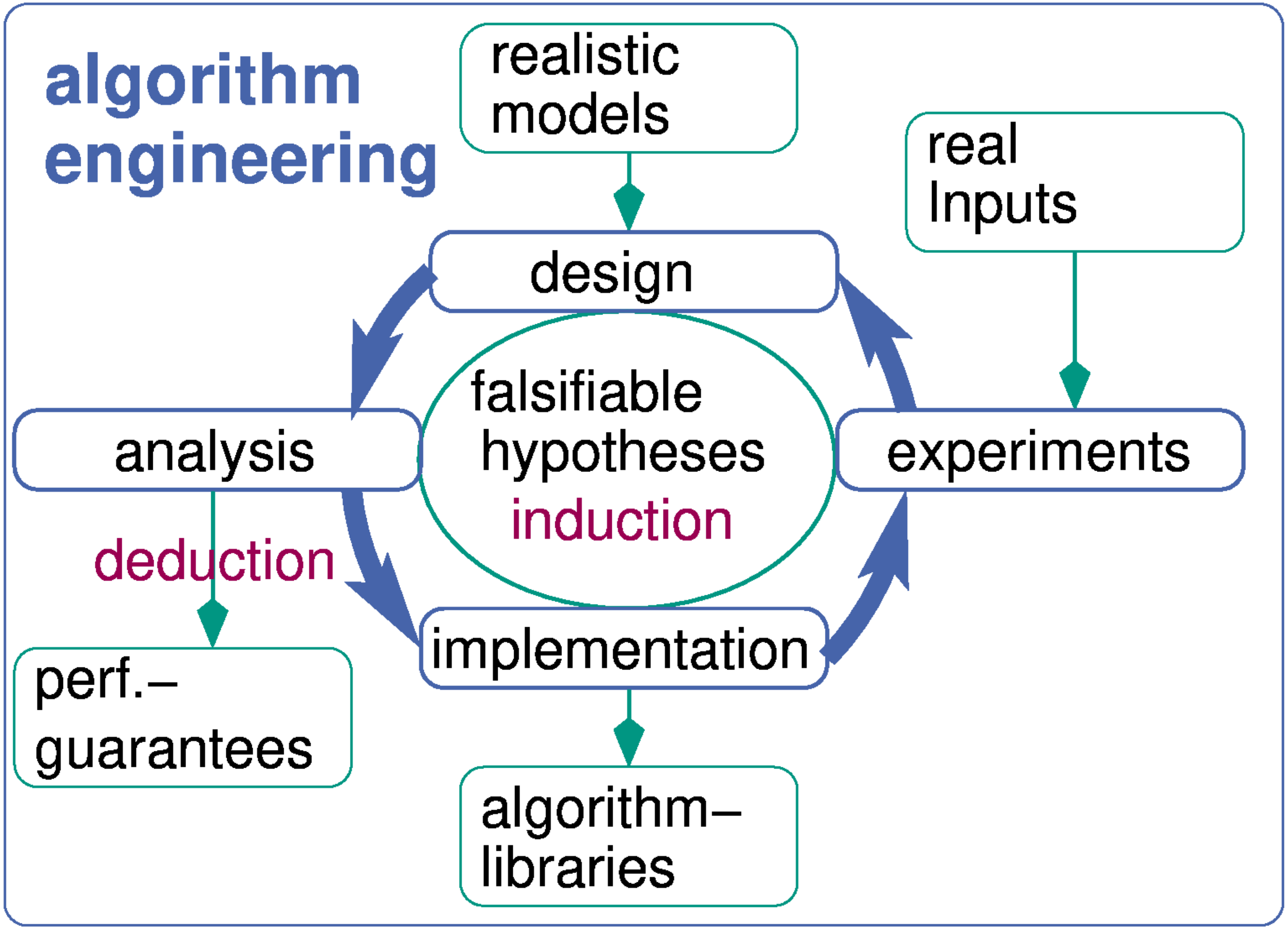
Algorithmics as Algorithm Engineering



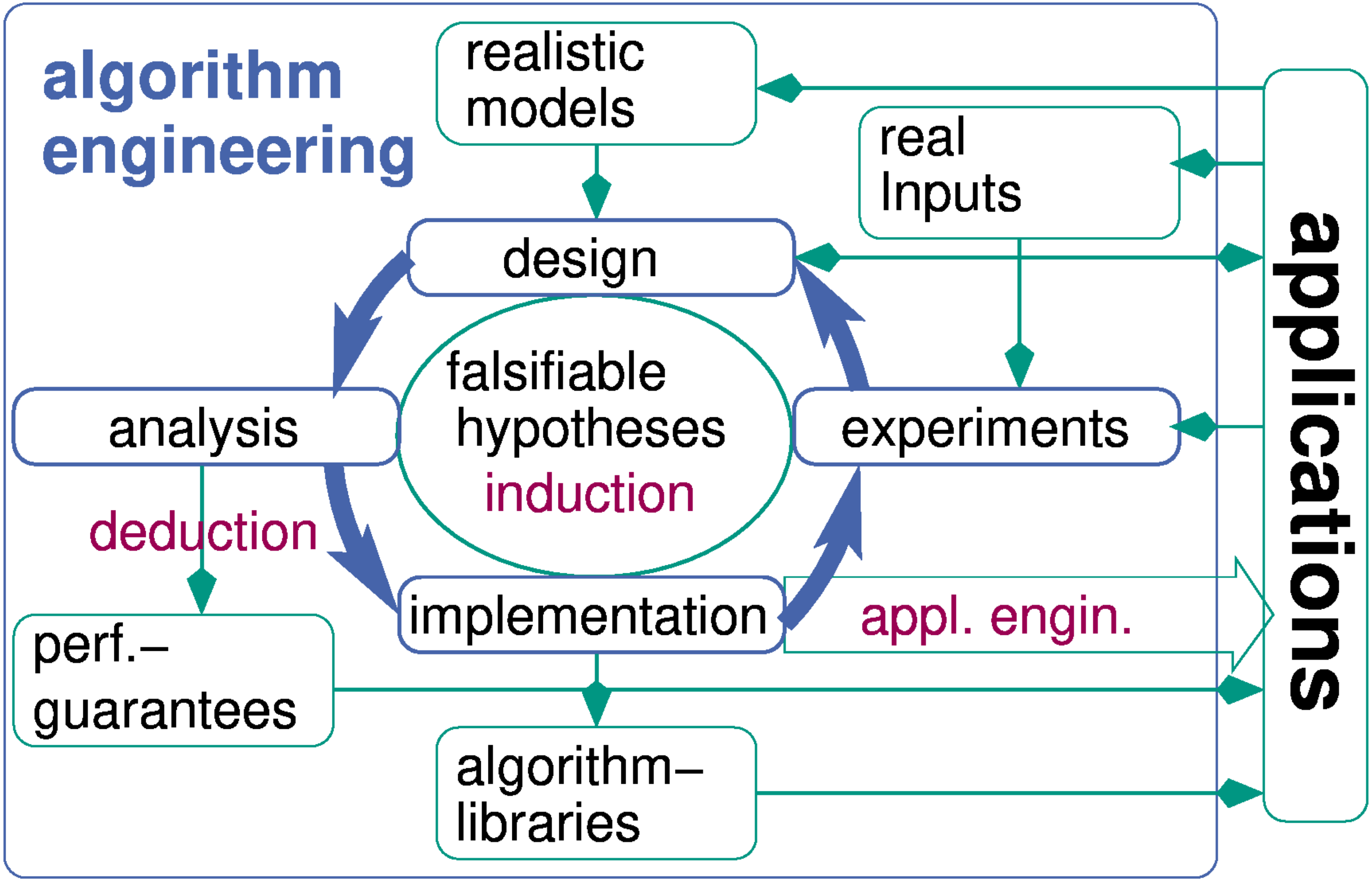
Algorithmics as Algorithm Engineering



Algorithmics as Algorithm Engineering



Algorithmics as Algorithm Engineering



Algorithm Engineering $\leftrightarrow$Algorithm Theory

Karlsruhe Institute of Technology

Conclusion:

- algorithm engineering is a wider view on algorithmics
  (but no revolution. None of the ingredients is really new)
- rich methodology
- better coupling to applications
- experimental algorithmics $\ll$ algorithm engineering
- algorithm theory $\subset$ algorithm engineering
- sometimes different theoretical questions
- algorithm theory may still yield the strongest, deepest and most persistent results within algorithm engineering

Theoretical Foundations

Algorithm Characterization

An algorithm can be characterized by:

- runtime behaviour
- (main) memory consumption
- I/O operations (e.g. hard drive)
- number and size of messages sent/received over network

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

$$T(n) := \ldots$$

Institute of Theoretical Informatics
Algorithmics Group

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

$$T(n) := \ldots$$

Examples

$m \leftarrow \frac{1}{2}\left(\mathcal{I}_0 + \mathcal{I}_{n-1}\right)$
return m

- $T(n) = 3$
- **Output:** undef.

Institute of Theoretical Informatics
Algorithmics Group

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

$$T(n) := \dots$$

Examples

Require: $\mathcal{I}$ sorted
  $m \leftarrow \frac{1}{2}\left(\mathcal{I}_0 + \mathcal{I}_{n-1}\right)$
  return m

- $T(n) = 3$
- **Output:** avg($\mathcal{I}$)

Institute of Theoretical Informatics
Algorithmics Group

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

$$T(n) := \ldots$$

Examples

Require: $\mathcal{I}$ sorted
  $m \leftarrow \frac{1}{2}\left(\mathcal{I}_0 + \mathcal{I}_{n-1}\right)$
  return m

- $T(n) = 3$
- **Output:** avg($\mathcal{I}$)

$a \leftarrow \infty, b \leftarrow 0$
for $i \in \mathcal{I}$ **do**
  if $i < a$ **then** $a \leftarrow i$
  if $i > b$ **then** $b \leftarrow i$
$m \leftarrow \frac{a+b}{2}$
return m

- $T(n) = 2n + 2$
- **Output:** avg($\mathcal{I}$)

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

$$T(n) := \dots$$

Examples

Require: $\mathcal{I}$ sorted
  $m \leftarrow \frac{1}{2}\left(\mathcal{I}_0 + \mathcal{I}_{n-1}\right)$
  return m

- $T(n) = 3$
- **Output:** avg($\mathcal{I}$)

for $i \in [0, |\mathcal{I}| - 1)$ **do**
  for $j \in [0, |\mathcal{I}| - i - 1)$ **do**
    if $\mathcal{I}_j > \mathcal{I}_{j+1}$ **then**
      swap($\mathcal{I}_j, \mathcal{I}_{j+1}$)
$m \leftarrow \frac{1}{2}\left(\mathcal{I}_0 + \mathcal{I}_{n-1}\right)$
return m

- $T(n) = 3n^2 + 3$
- **Output:** avg($\mathcal{I}$)
- **Side effect:** sorted $\mathcal{I}$

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

$$T(n) := \dots$$

Examples

Require: $\mathcal{I}$ sorted
  $m \leftarrow \frac{1}{2}\left(\mathcal{I}_0 + \mathcal{I}_{n-1}\right)$
  return m

- $T(n) = 3$
- **Output:** avg($\mathcal{I}$)

for $i \in [0, |\mathcal{I}| - 1)$ **do**
  for $j \in [0, |\mathcal{I}| - i - 1)$ **do**
    if $\mathcal{I}_j > \mathcal{I}_{j+1}$ **then**
      swap($\mathcal{I}_j, \mathcal{I}_{j+1}$)
$m \leftarrow \mathcal{I}_{n-1}$
for $i \in \mathcal{I}$ **do**
  $\mathcal{I}_i \leftarrow \frac{\mathcal{I}_i}{m}$

- $T(n) = 3n^2 + 2n + 1$
- **Side effect:** norm., sort. $\mathcal{I}$

Bachmann-Landau Notation

Consider $T(n) = 3n^2 + 2n + 1$:

- counting constant factors is tidious and can be architecture-dependant
- n^2 term clearly dominates lower order terms for sufficiently large n

Institute of Theoretical Informatics
Algorithmics Group

Bachmann-Landau Notation

Consider $T(n) = 3n^2 + 2n + 1$:

- counting constant factors is tidious and can be architecture-dependant
- n^2 term clearly dominates lower order terms for sufficiently large n

Enter Big-O notation

For upper bounds: $f(n) \in \mathrm{O}(g(n))$

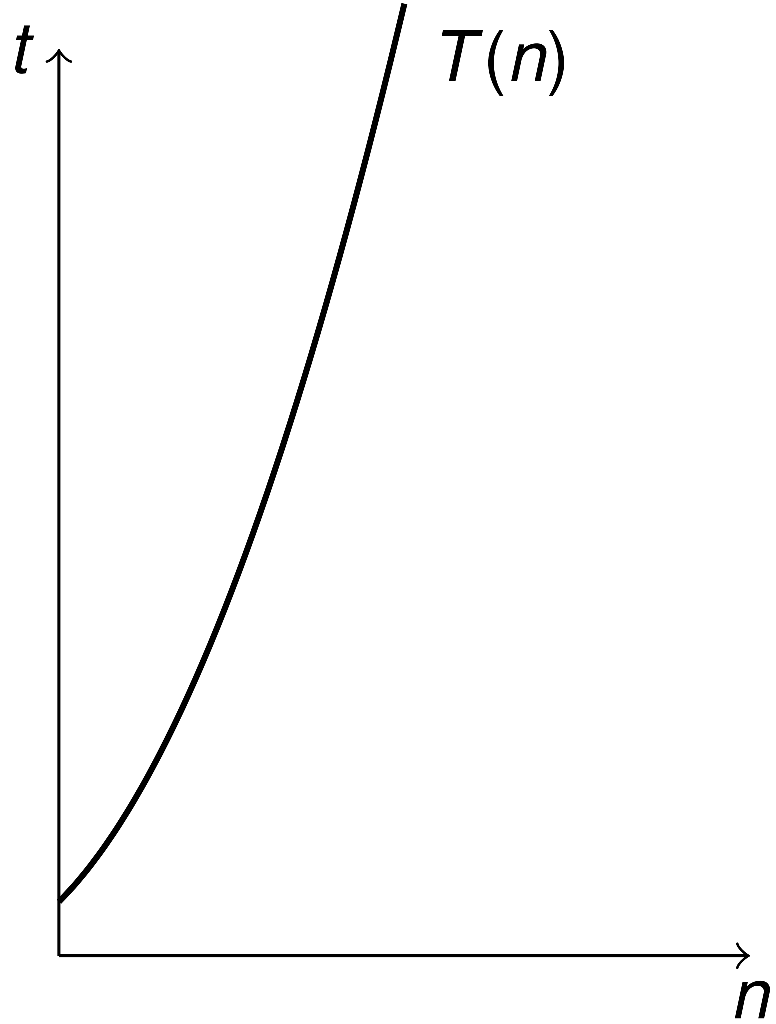
- $|f|$ is bounded above by g asymptotically (up to a constant factor)
- "$g(n)$ grows at least as fast as $f(n)$"
- Formally,

$$\exists k > 0 : \exists n_0 : \forall n > n_0 : |f(n)| \leq k \cdot g(n)$$

Institute of Theoretical Informatics
Algorithmics Group

Bachmann-Landau Notation

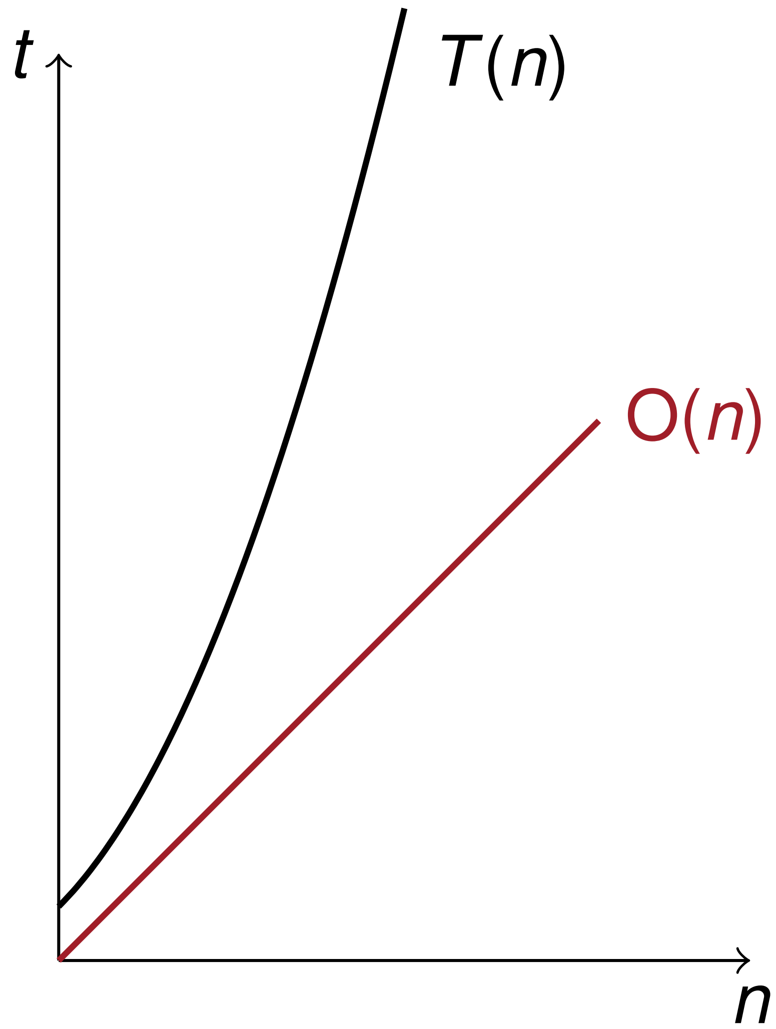
Given $T(n)$:



Bachmann-Landau Notation

Given $T(n)$:

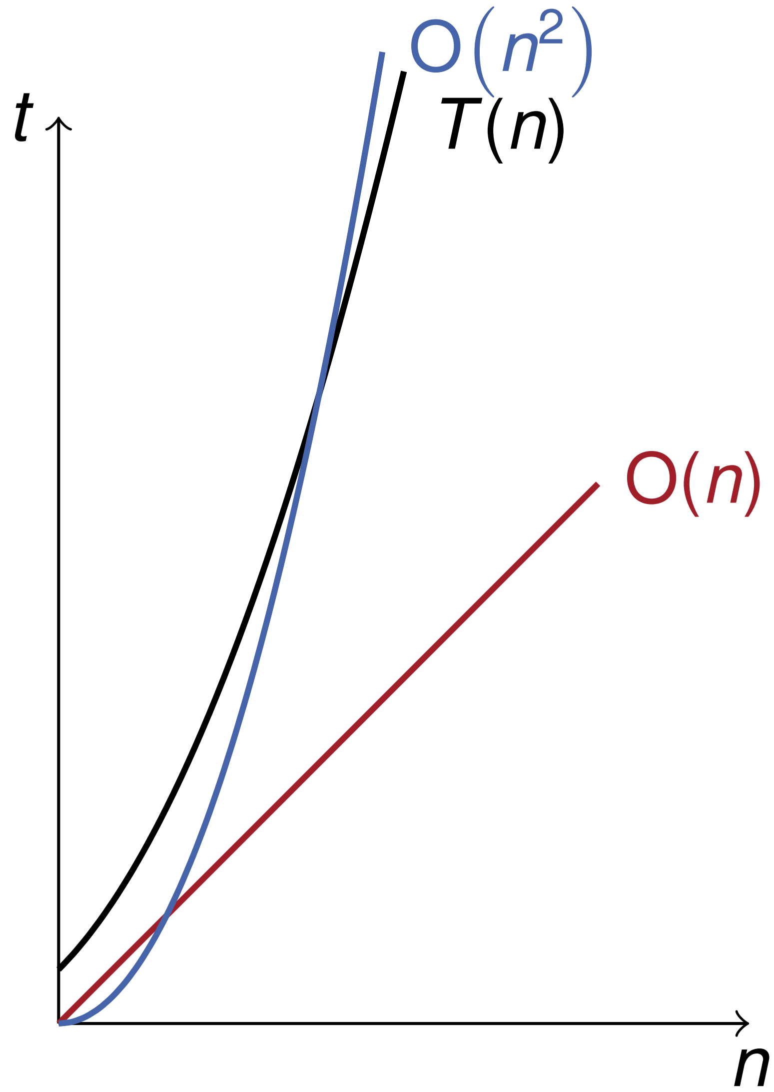
- $T(n) \notin O(n)$



Bachmann-Landau Notation

Given $T(n)$:

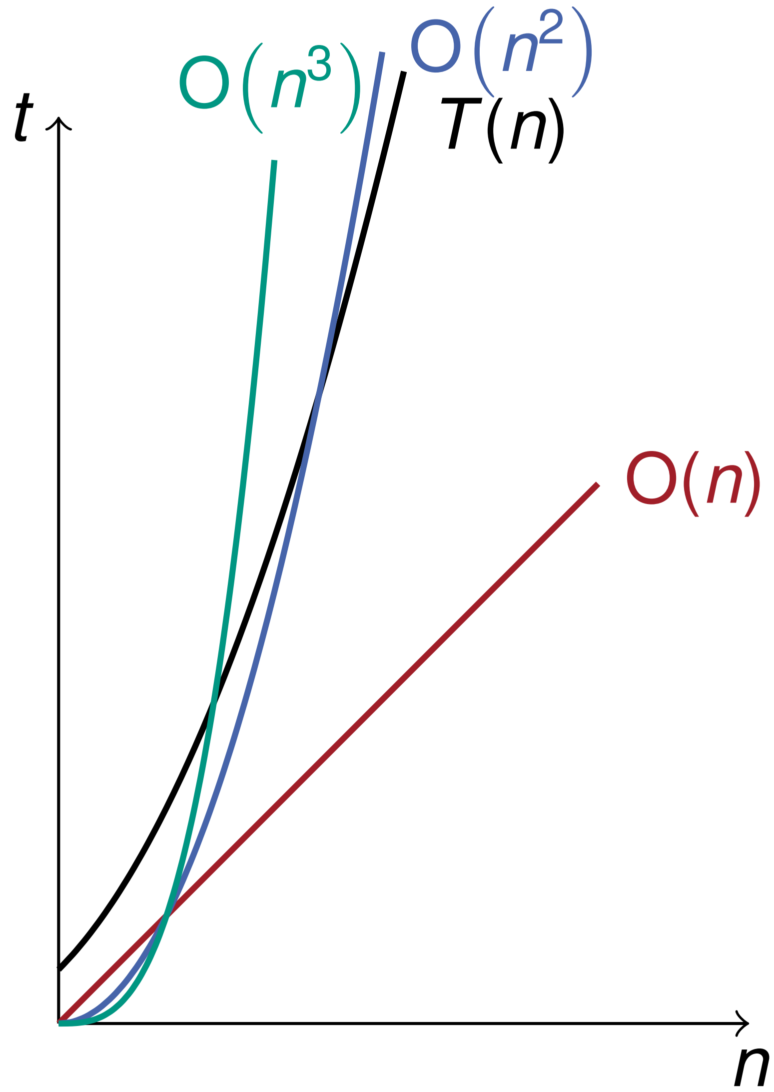
- $T(n) \notin O(n)$
- $T(n) \in O\left(n^2\right)$



Bachmann-Landau Notation

Given $T(n)$:

- $T(n) \notin O(n)$
- $T(n) \in O\left(n^2\right)$
- $T(n) \in O\left(n^3\right)$

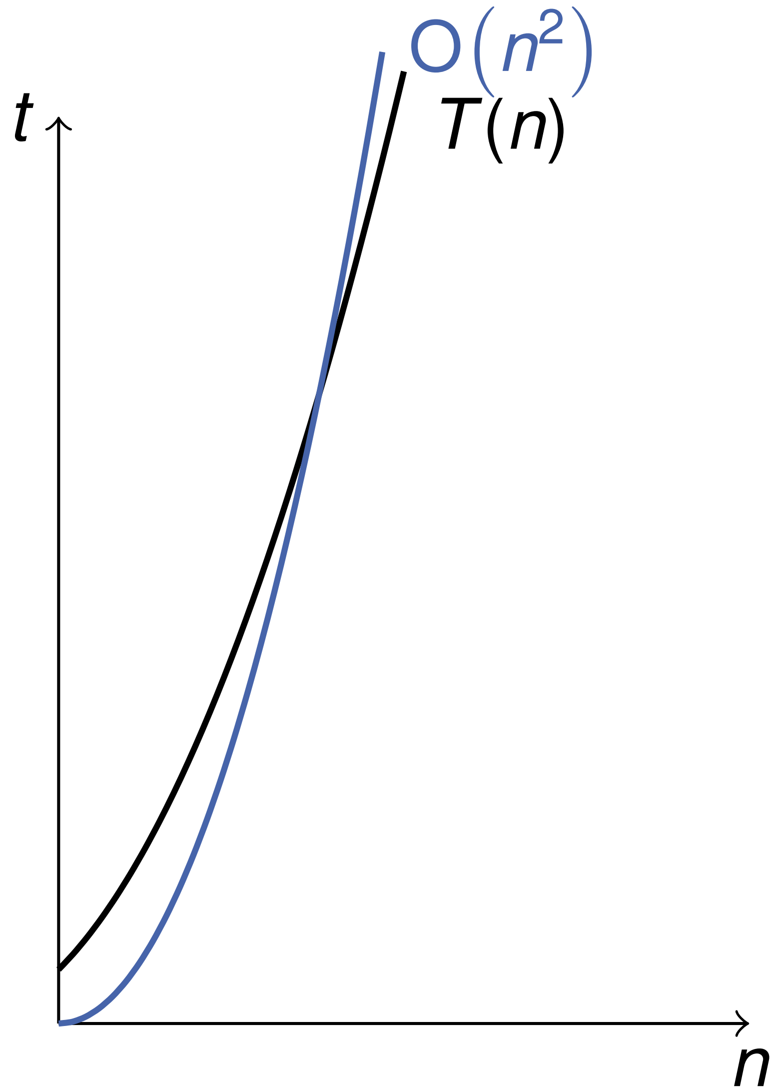


Bachmann-Landau Notation

Given $T(n)$:

- $T(n) \notin O(n)$
- $T(n) \in O\left(n^2\right)$
- $T(n) \in O\left(n^3\right)$

Tight bounds are preferred



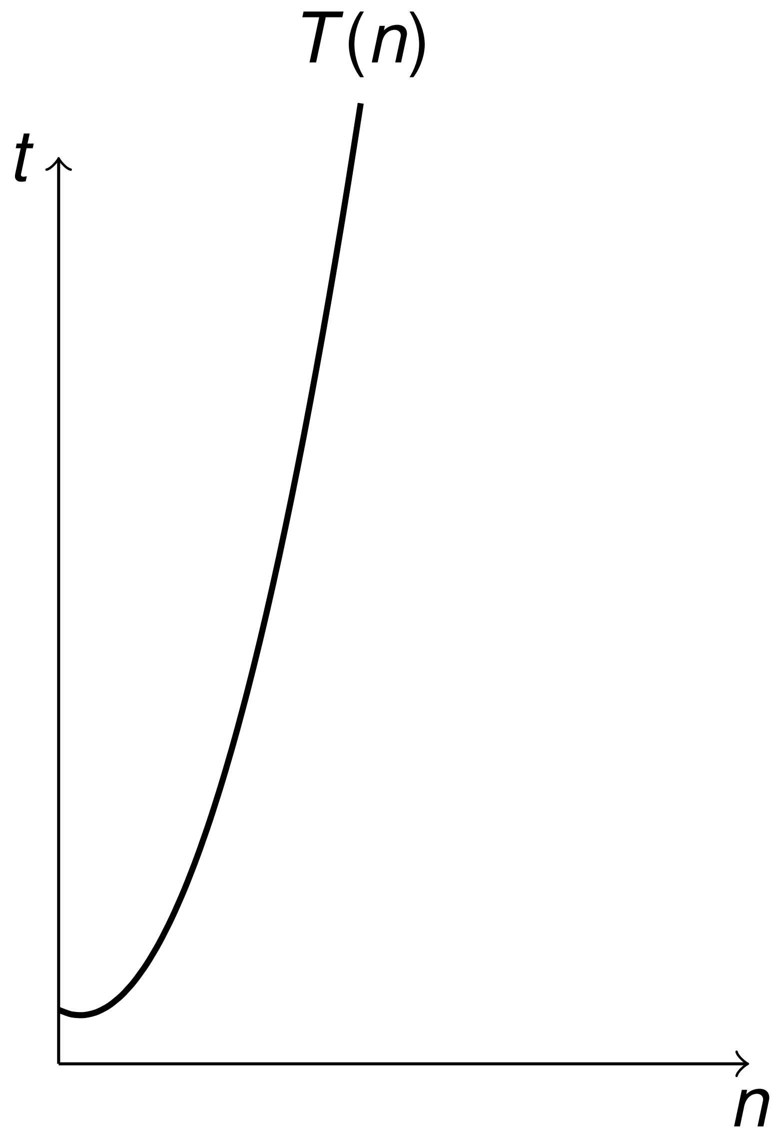
Bachmann-Landau Notation

For lower bounds: $f(n) \in \Omega(g(n))$

- $|f|$ is bounded below by g asymptotically (up to a constant factor)
- "$g(n)$ grows at most as fast as $f(n)$"
- Formally,
$$\exists k > 0 : \exists n_0 : \forall n > n_0 : f(n) \geq k \cdot g(n)$$

Bachmann-Landau Notation

Given $T(n)$:

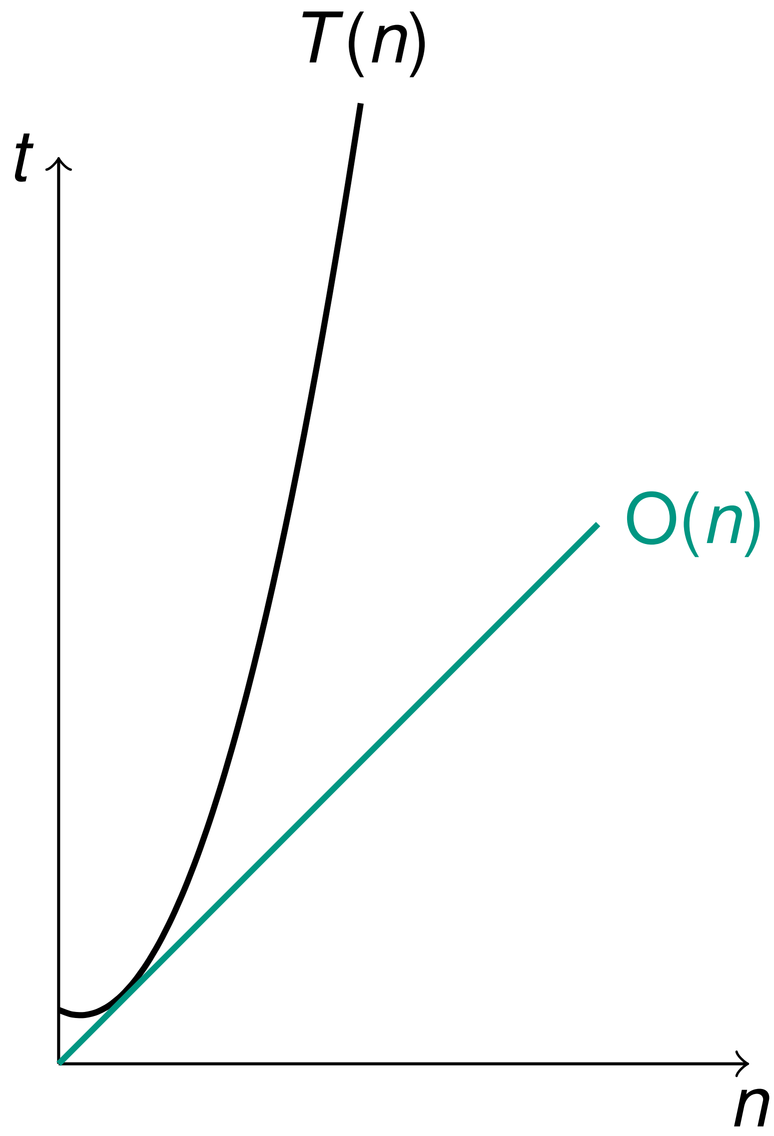


Institute of Theoretical Informatics
Algorithmics Group

Bachmann-Landau Notation

Given $T(n)$:

- $T(n) \in \Omega(n)$

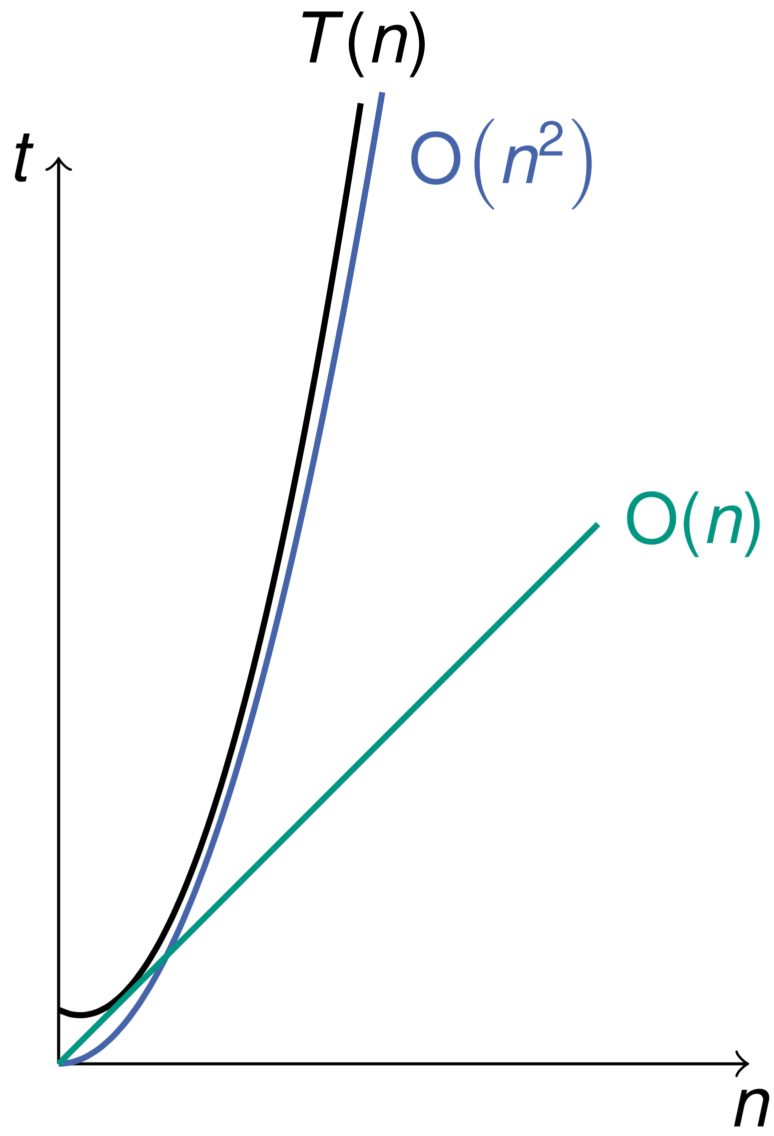


Bachmann-Landau Notation

Karlsruhe Institute of Technology

Given $T(n)$:

- $T(n) \in \Omega(n)$
- $T(n) \in \Omega\left(n^2\right)$

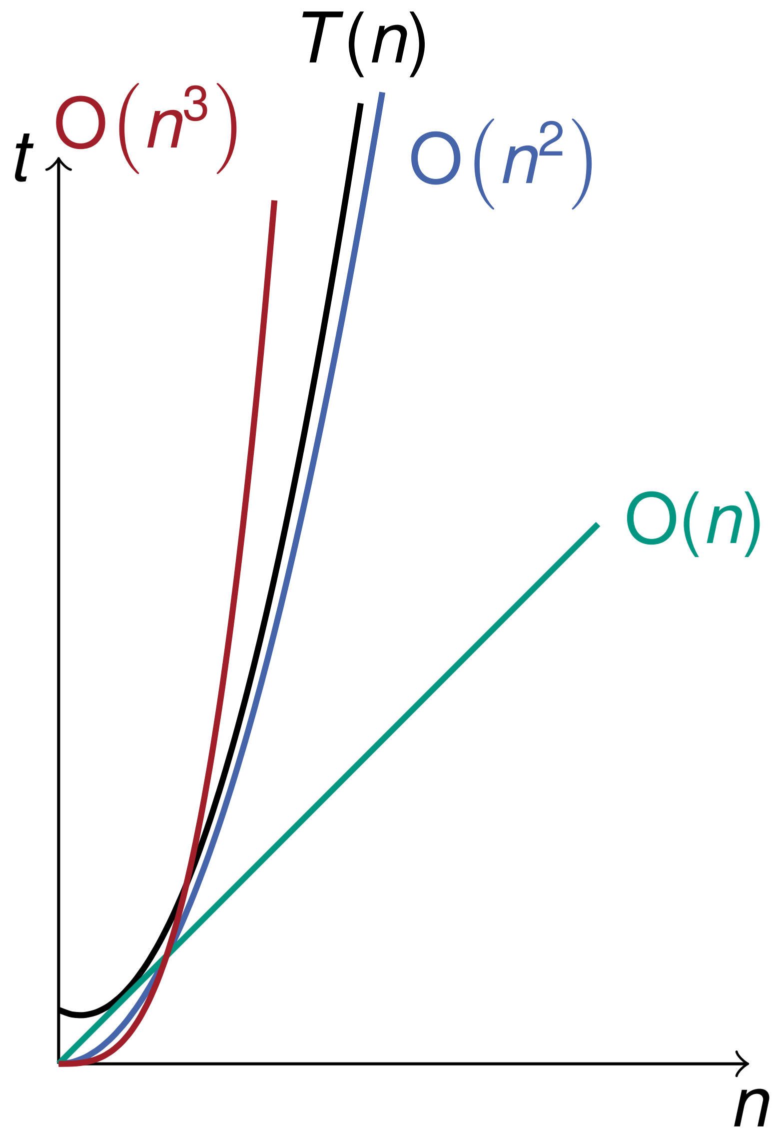


Institute of Theoretical Informatics
Algorithmics Group

Bachmann-Landau Notation

Given $T(n)$:

- $T(n) \in \Omega(n)$
- $T(n) \in \Omega\left(n^2\right)$
- $T(n) \notin \Omega\left(n^3\right)$

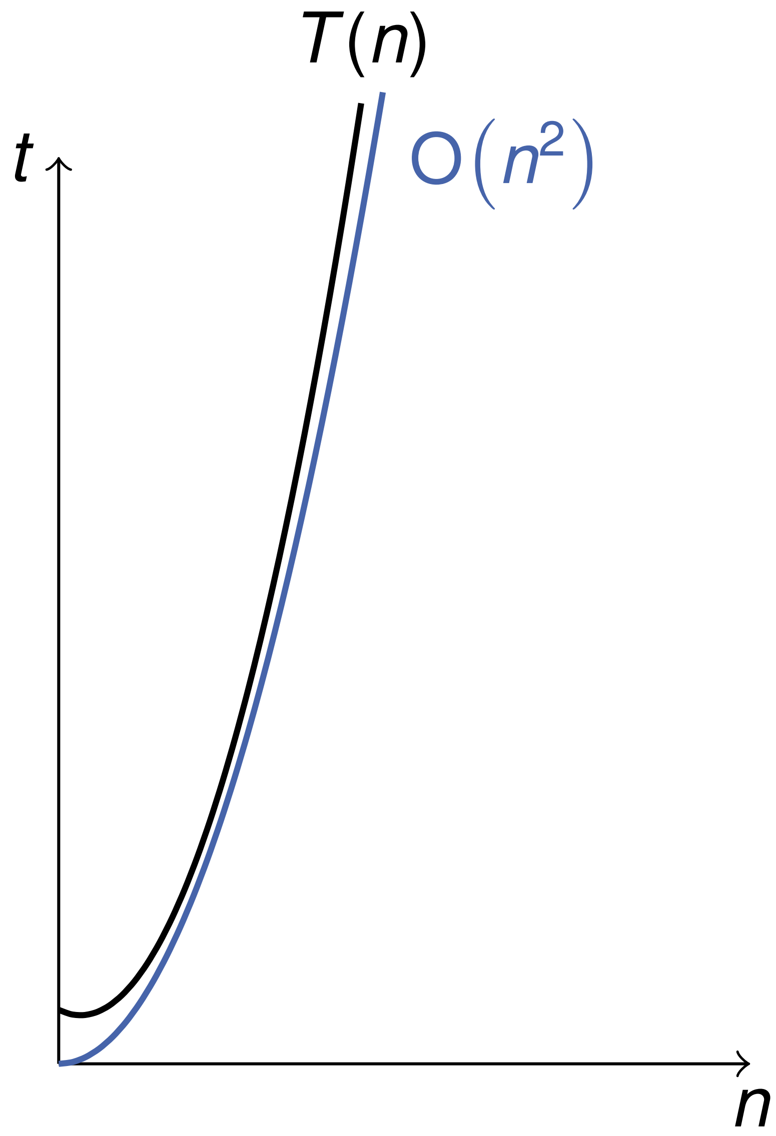


Bachmann-Landau Notation

Given $T(n)$:

- $T(n) \in \Omega(n)$
- $T(n) \in \Omega\left(n^2\right)$
- $T(n) \notin \Omega\left(n^3\right)$

Tight bounds are preferred



Bachmann-Landau Notation

For tight bounds: $f(n) \in \Theta(g(n))$

- $|f|$ is bounded both above and below by g asymptotically
- "$g(n)$ grows at as fast as $f(n)$"
- Formally,

$$\exists k_1, k_2 > 0 : \exists n_0 : \forall n > n_0 : k_1 \cdot g(n) \leq f(n) \geq k_2 \cdot g(n)$$

- $f(n) \in O(g(n))$ & $f(n) \in \Omega(g(n)) \Leftrightarrow f(n) \in \Theta(g(n))$

Institute of Theoretical Informatics
Algorithmics Group

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

sorted $\leftarrow$ **true**, $i \leftarrow 0$
while $i < |\mathcal{I}| - 1$ & sorted **do**
  if $\mathcal{I}_i > \mathcal{I}_{i+1}$ **then**
    sorted $\leftarrow$ **false**
  inc(i)
if $\neg$sorted **then**
  for $i \in [0, |\mathcal{I}| - 1)$ **do**
    for $j \in [0, |\mathcal{I}| - i - 1)$ **do**
      if $\mathcal{I}_j > \mathcal{I}_{j+1}$ **then**
        swap($\mathcal{I}_j, \mathcal{I}_{j+1}$)

Institute of Theoretical Informatics
Algorithmics Group

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

sorted $\leftarrow$ **true**, $i \leftarrow 0$
while $i < |\mathcal{I}| - 1$ & sorted **do**
    if $\mathcal{I}_i > \mathcal{I}_{i+1}$ **then**
        sorted $\leftarrow$ **false**
    inc(i)
if $\neg$sorted **then**
    for $i \in [0, |\mathcal{I}| - 1)$ **do**
        for $j \in [0, |\mathcal{I}| - i - 1)$ **do**
            if $\mathcal{I}_j > \mathcal{I}_{j+1}$ **then**
                swap($\mathcal{I}_j, \mathcal{I}_{j+1}$)

- sorted input:

$$\mathcal{I}_{\text{sorted}} = \{1, 2, 3, 4, 5, 6\}$$
$$T(n) = 2n + 2 \in O(n)$$

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

sorted $\leftarrow$ **true**, $i \leftarrow 0$
while $i < |\mathcal{I}| - 1$ & sorted **do**
  if $\mathcal{I}_i > \mathcal{I}_{i+1}$ **then**
    sorted $\leftarrow$ **false**
  inc(i)
if $\neg$sorted **then**
  for $i \in [0, |\mathcal{I}| - 1)$ **do**
    for $j \in [0, |\mathcal{I}| - i - 1)$ **do**
      if $\mathcal{I}_j > \mathcal{I}_{j+1}$ **then**
        swap($\mathcal{I}_j, \mathcal{I}_{j+1}$)

- sorted input:

$$\mathcal{I}_{\text{sorted}} = \{1, 2, 3, 4, 5, 6\}$$
$$T(n) = 2n + 2 \in O(n)$$

- descending input:

$$\mathcal{I}_{\text{desc}} = \{6, 5, 4, 3, 2, 1\}$$
$$T(n) = 3n^2 + 5 \in O\left(n^2\right)$$

Institute of Theoretical Informatics
Algorithmics Group

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends **only** on the size $|\mathcal{I}| =: n$

```
sorted ← true, i ← 0
while i < |I| − 1 & sorted do
    if I_i > I_{i+1} then
        sorted ← false
    inc(i)
if ¬sorted then
    for i ∈ [0, |I| − 1) do
        for j ∈ [0, |I| − i − 1) do
            if I_j > I_{j+1} then
                swap(I_j, I_{j+1})
```

- sorted input:

$$\mathcal{I}_{\text{sorted}} = \{1, 2, 3, 4, 5, 6\}$$
$$T(n) = 2n + 2 \in O(n)$$

- descending input:

$$\mathcal{I}_{\text{desc}} = \{6, 5, 4, 3, 2, 1\}$$
$$T(n) = 3n^2 + 5 \in O\left(n^2\right)$$

- almost sorted input:

$$\mathcal{I}_{\text{worst}} = \{1, 2, 3, 4, 6, 5\}$$
$$T(n) = 3n^2 + 2n + 2$$
$$\in O\left(n^2 + n\right) \in O\left(n^2\right)$$

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

To characterize an algorithm in theory:

- consider the worst case input
- determine tight upper bounds

Institute of Theoretical Informatics
Algorithmics Group

Algorithm Characterization

Given input $\mathcal{I}$, we assume the runtime depends only on the size $|\mathcal{I}| =: n$

To characterize an algorithm in theory:

- consider the worst case input
- determine tight upper bounds

To characterize an algorithm in practice:

- consider the instances at hand, often average case inputs
- determine bounds for the expected running time

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

Decision Problem:
Given an input $\mathcal{I}$, decide whether it belongs to a well-defined set $\mathbb{M}$.

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

Decision Problem:
Given an input $\mathcal{I}$, decide whether it belongs to a well-defined set $\mathbb{M}$.

Example: Boolean Satisfiability Problem (SAT)

Given a propositional logic formula

$$\phi\left[\mathbf{X}, \{\vee, \wedge, \neg\}\right] \text{ with variables } \mathbf{X} = \{x_1, x_2, \ldots, x_n\},$$

is there an assigment $\chi : \mathbf{X} \to \{\textbf{true}, \textbf{false}\}^n$ such that ϕ is satisfied?

Problem Characterization

In general we consider algorithms for two kinds of problems:

Decision Problem:
Given an input $\mathcal{I}$, decide whether it belongs to a well-defined set $\mathbb{M}$.

Example: Boolean Satisfiability Problem (SAT)

Given a propositional logic formula

$$\phi\left[\mathbf{X}, \{\vee, \wedge, \neg\}\right] \text{ with variables } \mathbf{X} = \{x_1, x_2, \ldots, x_n\},$$

is there an assigment $\chi : \mathbf{X} \to \{\textbf{true}, \textbf{false}\}^n$ such that ϕ is satisfied?

$$\phi_1 := (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

Problem Characterization

In general we consider algorithms for two kinds of problems:

Decision Problem:
Given an input $\mathcal{I}$, decide whether it belongs to a well-defined set $\mathbb{M}$.

Example: Boolean Satisfiability Problem (SAT)

Given a propositional logic formula

$$\phi\left[\mathbf{X}, \{\vee, \wedge, \neg\}\right] \text{ with variables } \mathbf{X} = \{x_1, x_2, \ldots, x_n\},$$

is there an assigment $\chi : \mathbf{X} \to \{\textbf{true}, \textbf{false}\}^n$ such that ϕ is satisfied?

$$\phi_1 := (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

$$\chi_1 := \mathbf{X} \to \mathbf{true}^n \quad \Rightarrow \quad \phi_1 \to \mathbf{true}$$

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

Decision Problem:

Given an input $\mathcal{I}$, decide whether it belongs to a well-defined set $\mathbb{M}$.

Example: Boolean Satisfiability Problem (SAT)

Given a propositional logic formula

$$\phi\left[\mathbf{X}, \{\vee, \wedge, \neg\}\right] \text{ with variables } \mathbf{X} = \{x_1, x_2, \ldots, x_n\},$$

is there an assigment $\chi : \mathbf{X} \to \{\textbf{true}, \textbf{false}\}^n$ such that ϕ is satisfied?

$$\begin{aligned}\phi_2 :=&(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)\\ &\wedge(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)\\ &\wedge(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)\end{aligned}$$

Problem Characterization

In general we consider algorithms for two kinds of problems:

Decision Problem:
Given an input $\mathcal{I}$, decide whether it belongs to a well-defined set $\mathbb{M}$.

Example: Boolean Satisfiability Problem (SAT)

Given a propositional logic formula

$$\phi\left[\mathbf{X}, \{\vee, \wedge, \neg\}\right] \text{ with variables } \mathbf{X} = \{x_1, x_2, \ldots, x_n\},$$

is there an assigment $\chi : \mathbf{X} \rightarrow \{\textbf{true}, \textbf{false}\}^n$ such that ϕ is satisfied?

$$\begin{aligned}\phi_2 := &(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \\ &\wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \\ &\wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)\end{aligned}$$

$\Rightarrow \phi_2$ not satisfiable e.g. $\chi_2 := \mathbf{X} \rightarrow \mathbf{true}^n$

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

Optimization Problem:
Given a set $\mathcal{L}$ of feasible solutions and cost function $f : \mathcal{L} \to \mathbb{R}$,
find $x^* \in \mathcal{L}$ such that

$$f(x^*) \leq f(x) \quad \forall x \in \mathcal{L}.$$

Problem Characterization

In general we consider algorithms for two kinds of problems:

Optimization Problem:
Given a set $\mathcal{L}$ of feasible solutions and cost function $f : \mathcal{L} \to \mathbb{R}$,
find $x^* \in \mathcal{L}$ such that

$$f(x^*) \leq f(x) \quad \forall x \in \mathcal{L}.$$

Example: Max-SAT
Given a propositional logic formula ϕ with variables $\mathbf{X}$,
which assigment χ maximizes the number of satisifed clauses $\#(\phi, \chi)$?

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

Optimization Problem:
Given a set $\mathcal{L}$ of feasible solutions and cost function $f : \mathcal{L} \to \mathbb{R}$,
find $x^* \in \mathcal{L}$ such that

$$f(x^*) \leq f(x) \quad \forall x \in \mathcal{L}.$$

Example: Max-SAT
Given a propositional logic formula ϕ with variables $\mathbf{X}$,
which assigment χ maximizes the number of satisifed clauses $\#(\phi, \chi)$?

$$\begin{aligned}\phi :=& (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)\\ &\wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)\\ &\wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)\end{aligned}$$

Problem Characterization

In general we consider algorithms for two kinds of problems:

Optimization Problem:
Given a set $\mathcal{L}$ of feasible solutions and cost function $f : \mathcal{L} \to \mathbb{R}$,
find $x^* \in \mathcal{L}$ such that

$$f(x^*) \leq f(x) \quad \forall x \in \mathcal{L}.$$

Example: Max-SAT
Given a propositional logic formula ϕ with variables $\mathbf{X}$,
which assigment χ maximizes the number of satisifed clauses $\#(\phi, \chi)$?

$$\begin{aligned} \phi :=& (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \\ & \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \\ & \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \end{aligned}$$

$$\chi_t : \mathbf{X} \to \mathbf{true}^n \quad \Rightarrow \quad \#(\phi, \chi_t) = 7$$

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

Optimization Problem:
Given a set $\mathcal{L}$ of feasible solutions and cost function $f : \mathcal{L} \to \mathbb{R}$,
find $x^* \in \mathcal{L}$ such that

$$f(x^*) \leq f(x) \quad \forall x \in \mathcal{L}.$$

Example: Max-SAT

Given a propositional logic formula ϕ with variables $\mathbf{X}$,
which assigment χ maximizes the number of satisifed clauses $\#(\phi, \chi)$?

$$\begin{aligned}\phi := &(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)\\ &\wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)\\ &\wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)\end{aligned}$$

$$\chi_f : \mathbf{X} \to \mathbf{false}^n \quad \Rightarrow \quad \#(\phi, \chi_f) = 7$$

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

Optimization Problem:
Given a set $\mathcal{L}$ of feasible solutions and cost function $f : \mathcal{L} \to \mathbb{R}$,
find $x^* \in \mathcal{L}$ such that

$$f(x^*) \leq f(x) \quad \forall x \in \mathcal{L}.$$

Example: Max-SAT
Given a propositional logic formula ϕ with variables $\mathbf{X}$,
which assigment χ maximizes the number of satisifed clauses $\#(\phi, \chi)$?

$$\begin{aligned}\phi :=&(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)\\ &\wedge(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)\\ &\wedge(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)\end{aligned}$$

$$\chi_m : \mathbf{X} \to \{\mathbf{true}, \mathbf{false}, \mathbf{false}\} \quad \Rightarrow \quad \#(\phi, \chi_m) = 7$$

Problem Characterization

In general we consider algorithms for two kinds of problems:

1. **Optimization Problem:**
   asks for the minimum cost solution $x^* \in \mathcal{L}$
2. **Optimal Value Problem:**
   asks for minimal cost function value $f(\cdot)$
3. **Decision Problem:**
   given a parameter $k \in \mathbb{R}$, asks $\exists x \in \mathcal{L}$ with $f(x) \leq k$?

Institute of Theoretical Informatics
Algorithmics Group

Problem Characterization

In general we consider algorithms for two kinds of problems:

solves

solves

1. **Optimization Problem:**
   asks for the minimum cost solution $x^* \in \mathcal{L}$
2. **Optimal Value Problem:**
   asks for minimal cost function value $f(\cdot)$
3. **Decision Problem:**
   given a parameter $k \in \mathbb{R}$, asks $\exists x \in \mathcal{L}$ with $f(x) \leq k$?

Institute of Theoretical Informatics
Algorithmics Group

Complexity Classes

Karlsruhe Institute of Technology

Complexity classes group problems of similar characteristics

- algorithm characeterized by its upper bound
- problem characterized by its lower bound, i.e.

  no possible algorithm can solve the problem faster than $T(n)$

- for many interesting problems lower bounds still unkown

$$\mathbf{P} \subset \mathbf{NP} \quad \textbf{?} \quad \mathbf{P} = \mathbf{NP}$$

Institute of Theoretical Informatics
Algorithmics Group

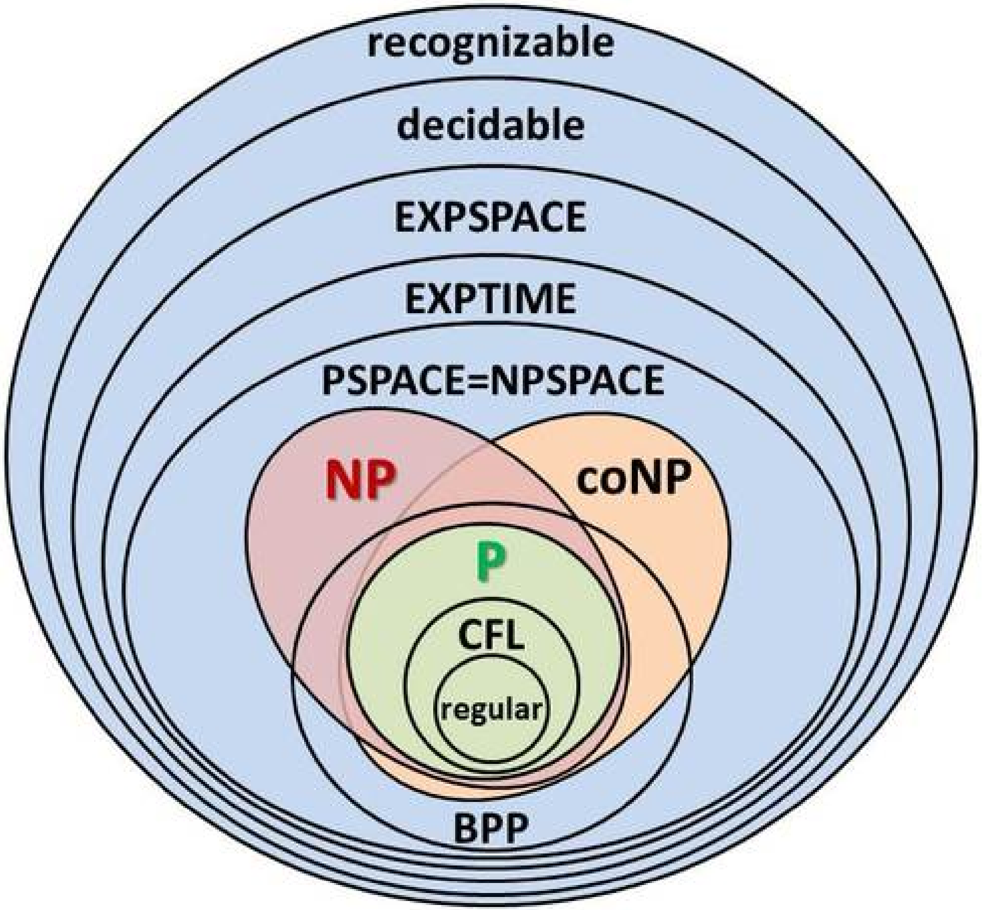
Complexity Classes

Complexity classes group problems of similar characteristics

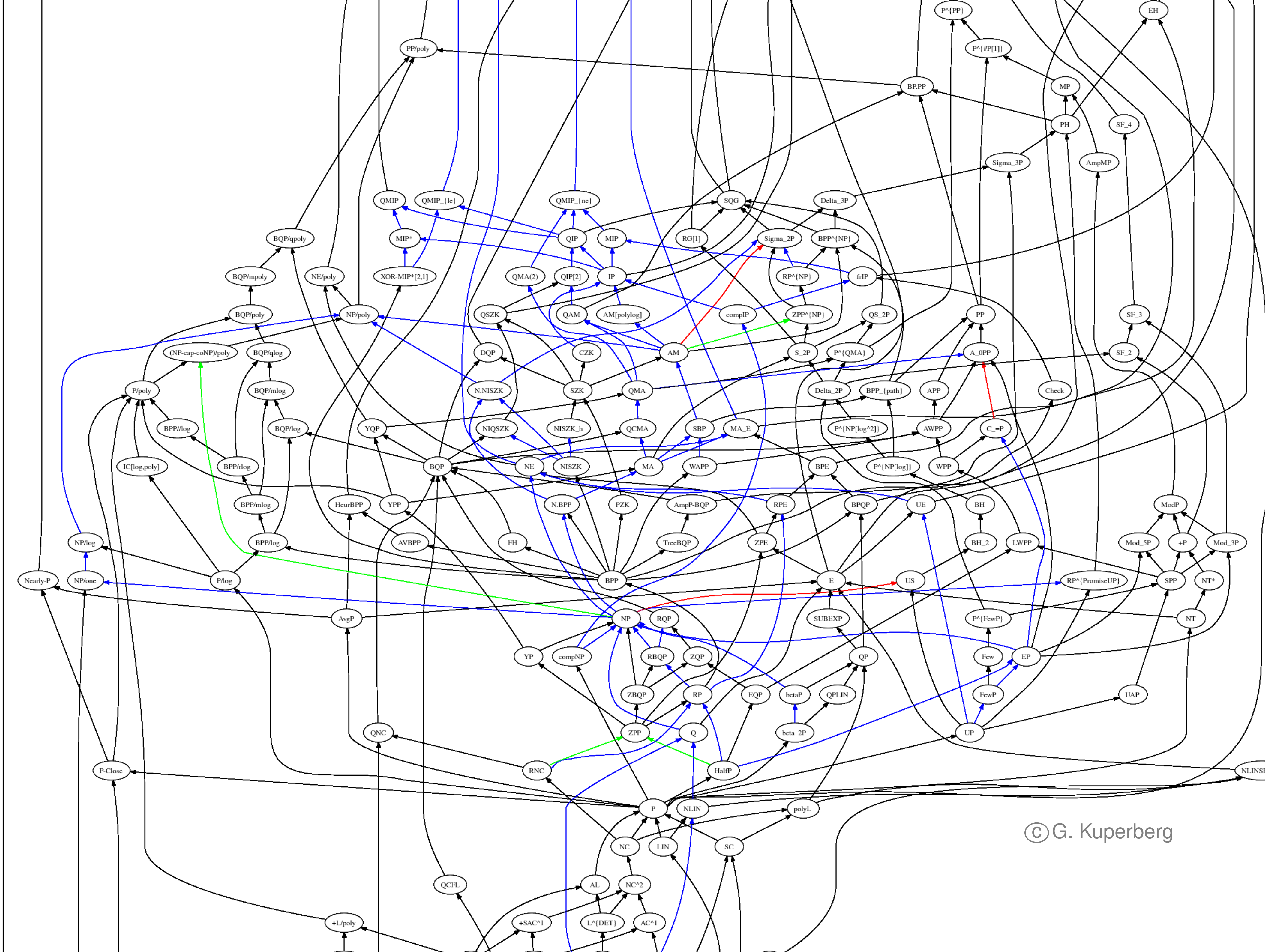
- algorithm chara
- problem charac

no possib ter than $T(n)$

- for many intere own



ⓒS. Raskhodnikova



P^{PP}
EH
PP/poly
P^{#P[1]}
BP.PP
MP
PH
SF_4
Sigma_3P
AmpMP
QMIP
QMIP_{le}
QMIP_{ne}
SQG
Delta_3P
BQP/qpoly
MIP*
QIP
MIP
RG[1]
Sigma_2P
BPP^{NP}
BQP/mpoly
NE/poly
XOR-MIP*[2,1]
QMA(2)
QIP[2]
IP
RP^{NP}
frIP
BQP/poly
NP/poly
QSZK
QAM
AM[polylog]
compIP
ZPP^{NP}
QS_2P
PP
SF_3
(NP-cap-coNP)/poly
BQP/qlog
DQP
CZK
AM
S_2P
P^{QMA}
A_0PP
SF_2
P/poly
BQP/mlog
N.NISZK
SZK
QMA
Delta_2P
BPP_{path}
APP
Check
BPP//log
BQP/log
YQP
NIQSZK
NISZK_h
QCMA
SBP
MA_E
P^{NP[log^2]}
AWPP
C_=P
IC[log,poly]
BPP/rlog
BQP
NE
NISZK
MA
WAPP
BPE
P^{NP[log]}
WPP
BPP/mlog
HeurBPP
YPP
N.BPP
PZK
AmpP-BQP
RPE
BPQP
UE
BH
ModP
NP/log
BPP/log
AVBPP
FH
TreeBQP
ZPE
BH_2
LWPP
Mod_5P
+P
Mod_3P
Nearly-P
NP/one
P/log
BPP
E
US
RP^{PromiseUP}
SPP
NT*
AvgP
NP
RQP
SUBEXP
P^{FewP}
NT
YP
compNP
RBQP
ZQP
QP
Few
EP
ZBQP
RP
EQP
betaP
QPLIN
FewP
UAP
QNC
ZPP
Q
beta_2P
UP
P-Close
RNC
HalfP
P
NLIN
polyL
NC
LIN
SC
QCFL
AL
NC^2
+L/poly
+SAC^1
L^{DET}
AC^1
©G. Kuperberg

The Good – The Bad

©PEA

Complexity Class P:
Problems decidable by a deterministic machine in polynomial time

$$T(n) \in \mathrm{O}\left(n^d\right) \quad \text{for constant } d.$$

Institute of Theoretical Informatics
Algorithmics Group

The Good – The Bad

© PEA

Complexity Class P:

Problems decidable by a deterministic machine in polynomial time

$$T(n) \in \mathrm{O}\left(n^{d}\right) \quad \text{for constant } d.$$

Examples:

- Circuit Value Problem (CVP)
- Linear programming
- Primality testing

The Good – The Bad

Complexity Class P:

Problems decidable by a deterministic machine in polynomial time

$$T(n) \in O\left(n^d\right) \quad \text{for constant } d.$$

Remarks:

- polynomial time algorithms are considered efficient
- in practice, algorithms $\in O\left(n^2\right)$ infeasible for large inputs
- algorithms $\in O(n \log n)$ desirable

The Good – The Bad

Complexity Class P:

Problems decidable by a deterministic machine in polynomial time

$$T(n) \in \mathrm{O}\left(n^d\right) \quad \text{for constant } d.$$

Complexity Class NP:

Problems decidable by a non-deterministic machine in polynomial time.

or

Set of decison problems with efficiently verifiable-proof for “yes” instances.

The Good – The Bad

Complexity Class P:
Problems decidable by a deterministic machine in polynomial time

$$T(n) \in \mathrm{O}\left(n^d\right) \quad \text{for constant } d.$$

Complexity Class NP:
Problems decidable by a non-deterministic machine in polynomial time.

or

Set of decison problems with efficiently verifiable-proof for "yes" instances.

Examples

- Boolean Satisfiability Problem (SAT)
- Knapsack Problem
- Subset sum problem

The Good – The Bad

$\mathbf{NP}$-complete:

Problem L is $\mathbf{NP}$-complete iff

1. $L \in \mathbf{NP}$
2. L is $\mathbf{NP}$-hard:
   every problem $G \in \mathbf{NP}$ can be reduced in polynomial time to L
   $\Leftrightarrow$ $\mathbf{NP}$-complete problem G can be reduced in polynomial time to L.

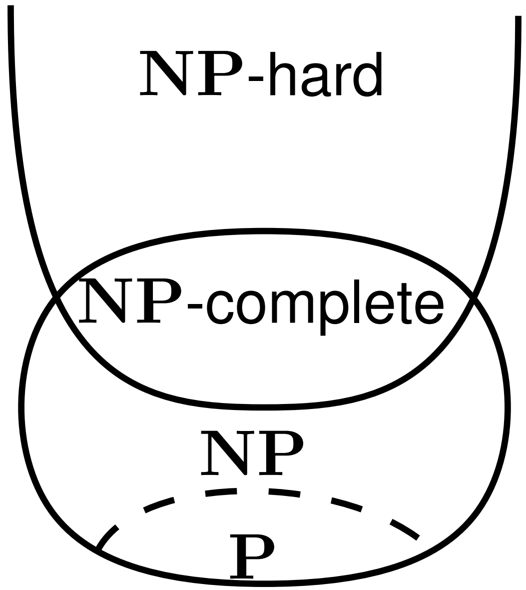
The Good – The Bad

© PEA

$\mathbf{NP}$-complete:

Problem L is $\mathbf{NP}$-complete iff

1. $L \in \mathbf{NP}$
2. L is $\mathbf{NP}$-hard:
   every problem $G \in \mathbf{NP}$ can be reduced in polynomial time to L
   $\Leftrightarrow$ $\mathbf{NP}$-complete problem G can be reduced in polynomial time to L.



The Bad

© PEA

Many interesting optimization problems are $\mathbf{NP}$-hard

Approximation algorithms:
Instead of exact solution x^*, compute approximate solution $\tilde{x}$ in polynomial time with provable goodness guarantee $f(n)$

$$\frac{\tilde{x}}{x^*} \leq f(n).$$

The Bad

Complexity Class APX:

Problems approximable to a constant factor c in polynomial time,

$$f(n) = c.$$

The Bad

Complexity Class APX:
Problems approximable to a constant factor c in polynomial time,

$$f(n) = c.$$

Complexity Class PTAS:
Problems approximable to any factor $1 + \epsilon$

$$f(n) = 1 + \epsilon \quad \forall \epsilon > 0,$$

with runtime polynomial in n but possibly exponential in $\frac{1}{\epsilon}$.

The Bad

©PEA

Complexity Class APX:
Problems approximable to a constant factor c in polynomial time,

$$f(n) = c.$$

Complexity Class PTAS:
Problems approximable to any factor $1 + \epsilon$

$$f(n) = 1 + \epsilon \quad \forall \epsilon > 0,$$

with runtime polynomial in n but possibly exponential in $\frac{1}{\epsilon}$.

Complexity Class FPTAS:
PTAS with runtime polynomial in n and $\frac{1}{\epsilon}$.

The Bad

Many interesting optimization problems are $\mathbf{NP}$-hard

Approximation algorithms:
Instead of exact solution x^*, compute approximate solution $\tilde{x}$ in polynomial time with provable goodness guarantee $f(n)$

$$\frac{\tilde{x}}{x^*} \leq f(n).$$

$\mathbf{FPTAS}$	$\subset$	$\mathbf{PTAS}$	$\subset$	$\mathbf{APX}$
Knapsack		Makespan scheduling		Bin packing

Institute of Theoretical Informatics
Algorithmics Group

The Ugly

©PEA

Some problems cannot be approximated efficiently

Institute of Theoretical Informatics
Algorithmics Group

The Ugly

Some problems cannot be approximated efficiently

Example: Minimum Set Cover

Given a universe $\mathbb{U} = \{1, 2, \ldots, n\}$ and a collection S of m subsets of $\mathbb{U}$, with $\bigcup_{s \in S} = \mathbb{U}$, find a minimal subfamily $C \subseteq S$ with $\bigcup_{c \in C} = \mathbb{U}$

Min set cover cannot be approximated to $(1 - o(1)) \cdot \log n$, unless $\mathbf{P} = \mathbf{NP}$.

Institute of Theoretical Informatics
Algorithmics Group

The Ugly

©PEA

Some problems cannot be approximated efficiently

Example: Minimum Set Cover

Given a universe $\mathbb{U} = \{1, 2, \ldots, n\}$ and a collection S of m subsets of $\mathbb{U}$, with $\bigcup_{s \in S} = \mathbb{U}$, find a minimal subfamily $C \subseteq S$ with $\bigcup_{c \in C} = \mathbb{U}$

Min set cover cannot be approximated to $(1 - o(1)) \cdot \log n$, unless $\mathbf{P} = \mathbf{NP}$.

- there can be polynomial time heuristics for these problems
- work good in practice, but without proven guarantee

Institute of Theoretical Informatics
Algorithmics Group

The Good – The Bad – The Ugly

© PEA

Good : $\leq O(n \log n)$

Goodish: $\geq O(n^2)$

Bad: $\mathbf{NP}$-hard

$\mathbf{APX}$

Ugly: $\mathbf{NP}$-hard

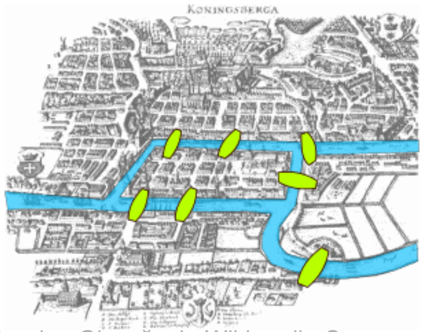
not $\mathbf{APX}$

Institute of Theoretical Informatics
Algorithmics Group

Graph Theory

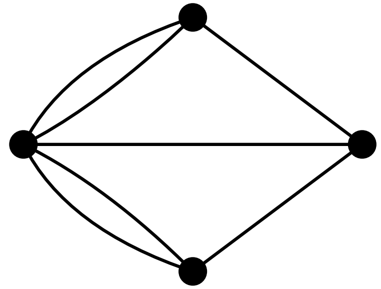
Graph Theory

- Foundation: 7 Bridges of Köngisberg (L. Euler, 1736)
  Problem: Walk through Königsberg crossing each bridge exacly once

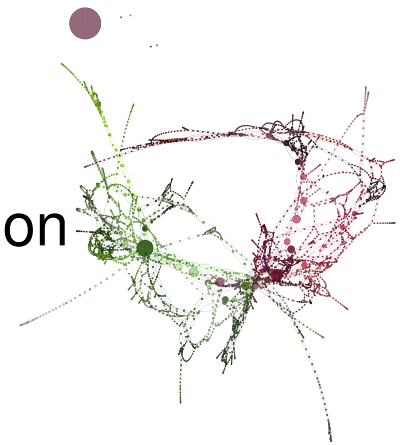


[Bogdan Giuşcă, via Wikimedia Commons]

$\rightsquigarrow$

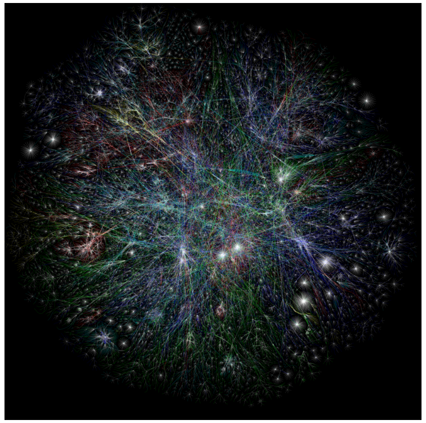


- Today: widely used to model **relationships** between **objects**
  - Social Networks
  - Transportation
  - Internet
  - Protein Interaction
  - ...



[sayasaya2011.wordpress.com]

[Martin Grandjean, via Wikimedia Commons]



[Barrett Lyon / The Opte Project]

Graphs: Notation & Definitions

Graph $G = (V, E)$

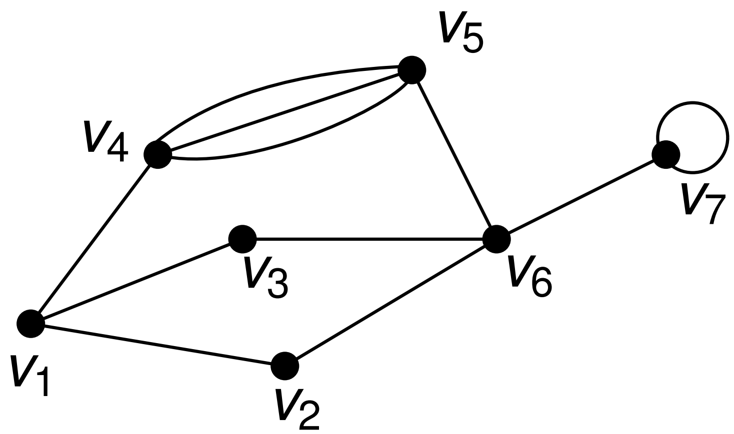
vertices edges

$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), \ldots\}$

$n = |V|$

$m = |E|$



Graphs: Notation & Definitions

Graph $G = (V, E)$

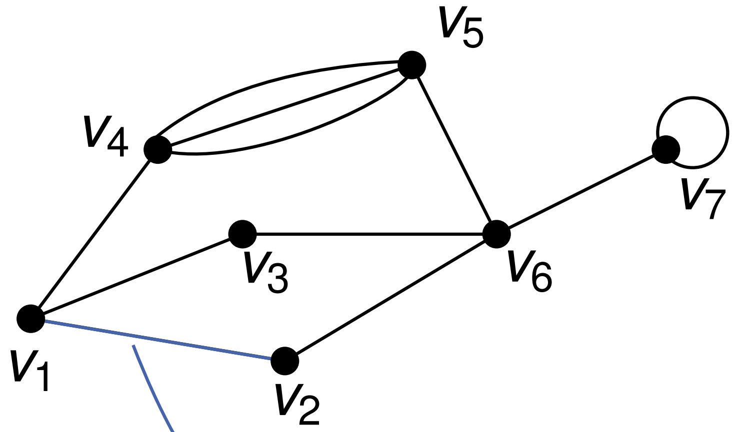
vertices — edges

$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), \ldots\}$

$n = |V|$

$m = |E|$



degree $d(v)$: # incident edges

$e_1 = (v_1, v_2)$ is incident to v_1, v_2

v_1 & v_2 are adjacent

Institute of Theoretical Informatics
Algorithmics Group

Graphs: Notation & Definitions

Graph $G = (V, E)$

vertices → V, edges → E

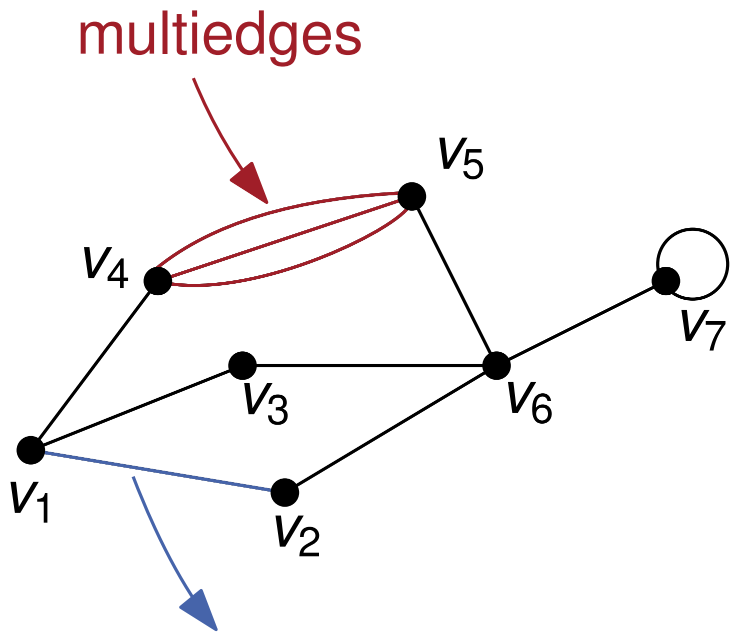
$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), \ldots\}$

$n = |V|$

$m = |E|$

degree $d(v)$: # incident edges



$e_1 = (v_1, v_2)$ is incident to v_1, v_2

v_1 & v_2 are adjacent

Graphs: Notation & Definitions

Graph $G = (V, E)$

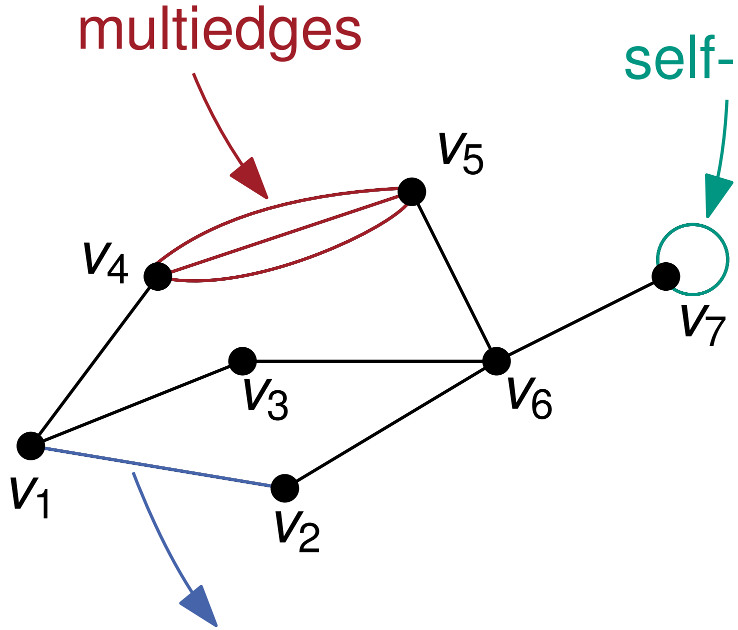
vertices — edges

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), \ldots\}$$

$$n = |V|$$

$$m = |E|$$



degree $d(v)$: # incident edges

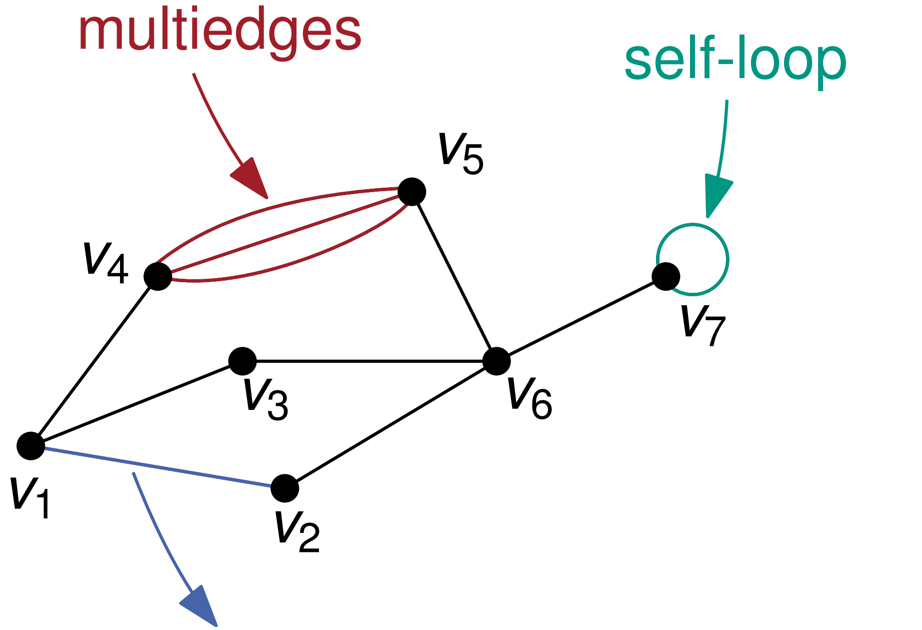
$e_1 = (v_1, v_2)$ is incident to v_1, v_2

v_1 & v_2 are adjacent

Graphs: Notation & Definitions

Graph $G = (V, E)$

vertices — edges



$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), \ldots\}$

$n = |V|$

$m = |E|$

$e_1 = (v_1, v_2)$ is incident to v_1, v_2

v_1 & v_2 are adjacent

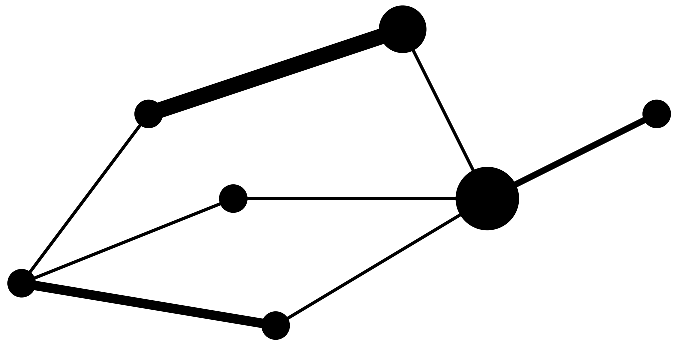
degree $d(v)$: # incident edges

simple graph: no self-loops & multiedges

Institute of Theoretical Informatics
Algorithmics Group

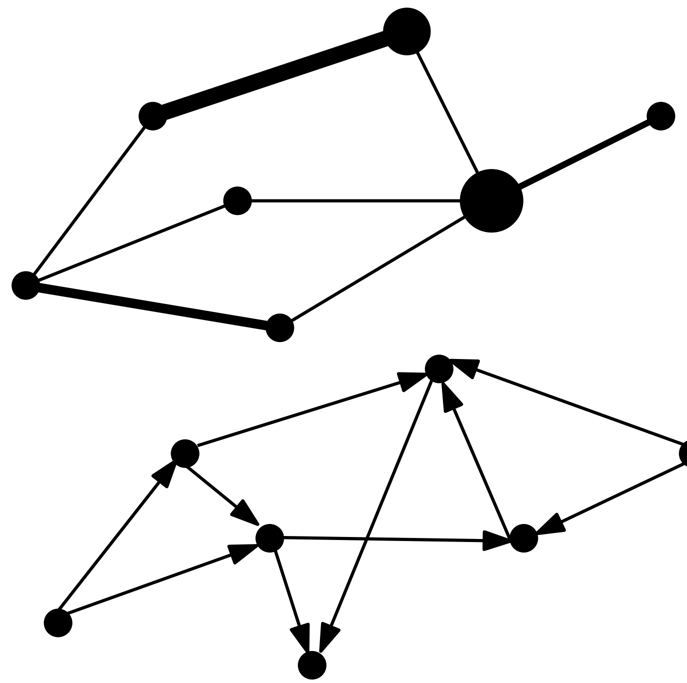
Graphs: Notation & Definitions

- **Weighted Graphs**:
  - vertex weights $c : V \to \mathbb{R}$
  - edge weights $\omega : E \to \mathbb{R}$



Graphs: Notation & Definitions

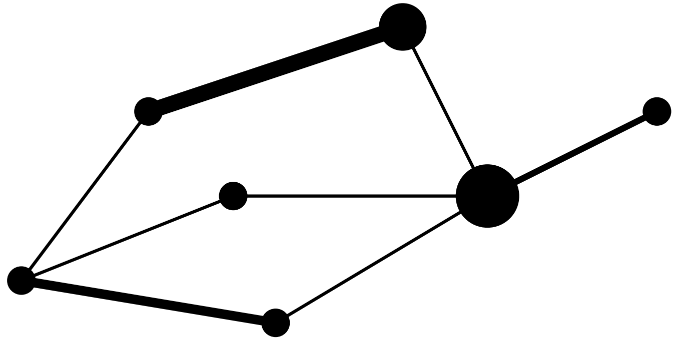
- **Weighted Graphs**:
  - vertex weights $c : V \to \mathbb{R}$
  - edge weights $\omega : E \to \mathbb{R}$



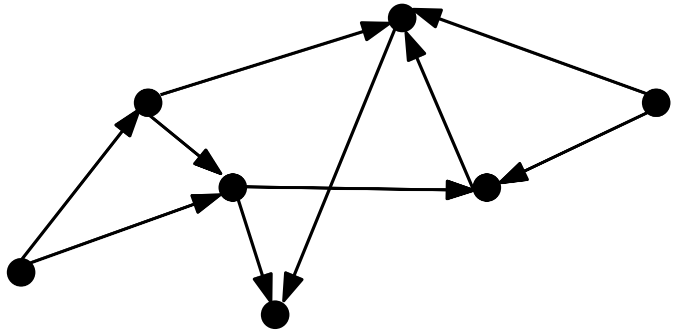
- **Directed Graphs**:
  - in-degree $d_{in}(v)$
  - out-degree $d_{out}(v)$

Graphs: Notation & Definitions

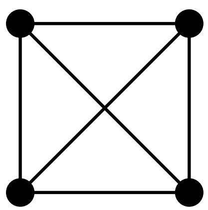
- **Weighted Graphs**:
  - vertex weights $c : V \to \mathbb{R}$
  - edge weights $\omega : E \to \mathbb{R}$



- **Directed Graphs**:
  - in-degree $d_{in}(v)$
  - out-degree $d_{out}(v)$

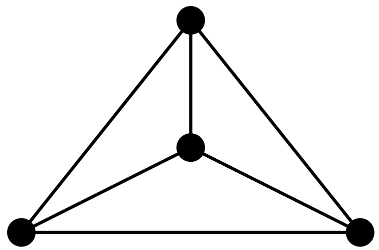


- **Planar Graphs**: can be drawn without edge crossings

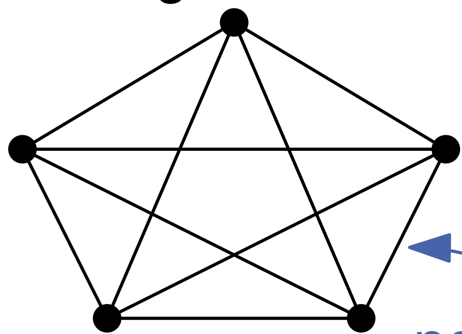


K_4

$\rightsquigarrow$



K_4

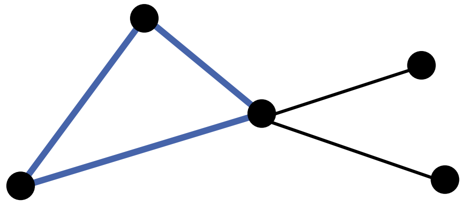


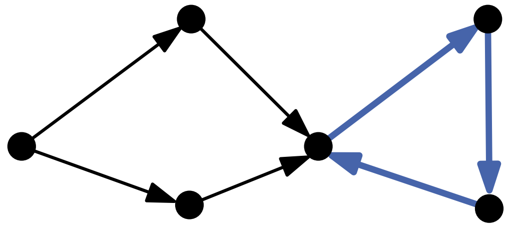
K_5

not planar

Graphs: Notation & Definitions

- **Cyclic Graphs**

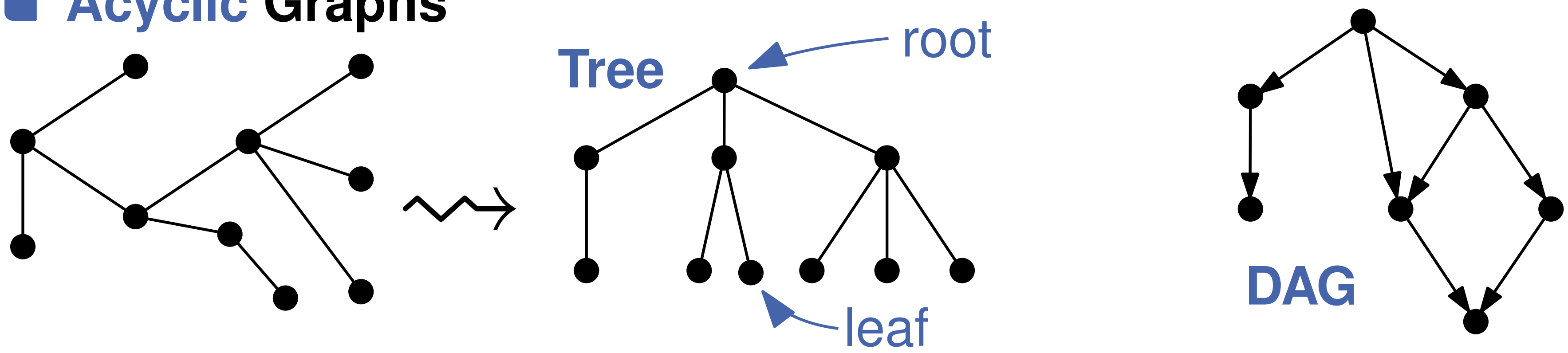




Graphs: Notation & Definitions

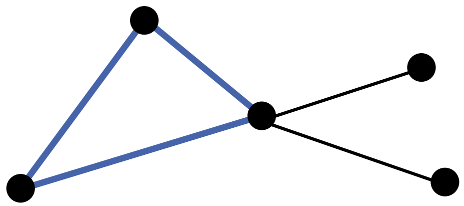
- **Cyclic Graphs**

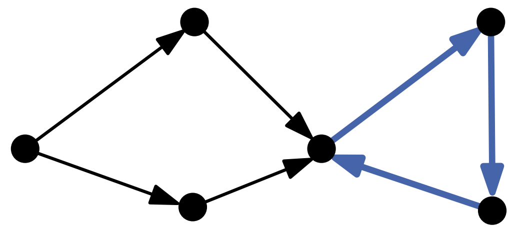
- **Acyclic Graphs**



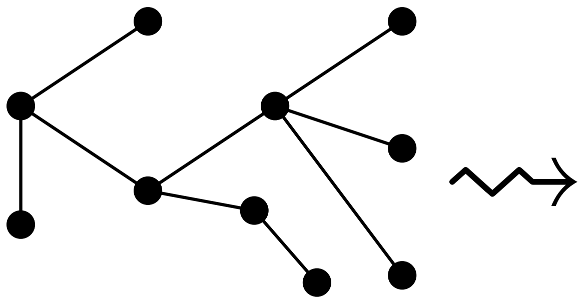
Graphs: Notation & Definitions

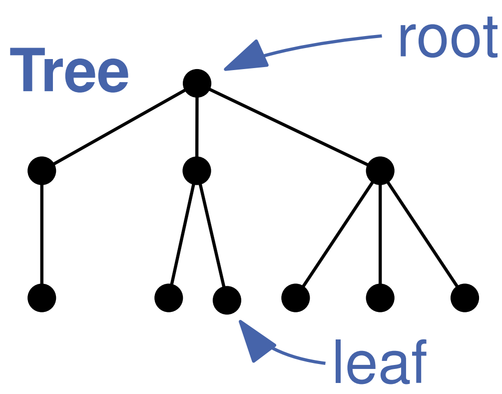
- **Cyclic Graphs**

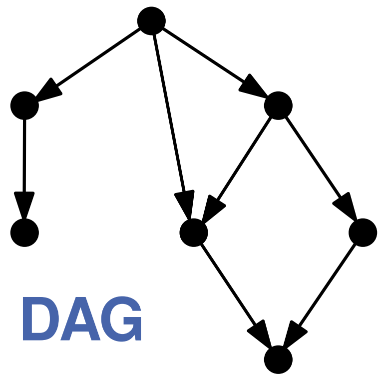




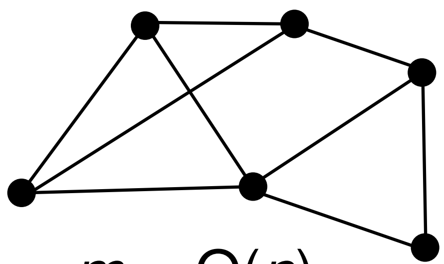
- **Acyclic Graphs**



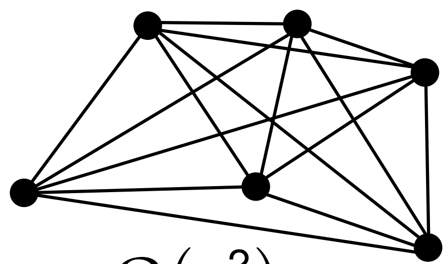




- **Sparse/Dense Graphs**



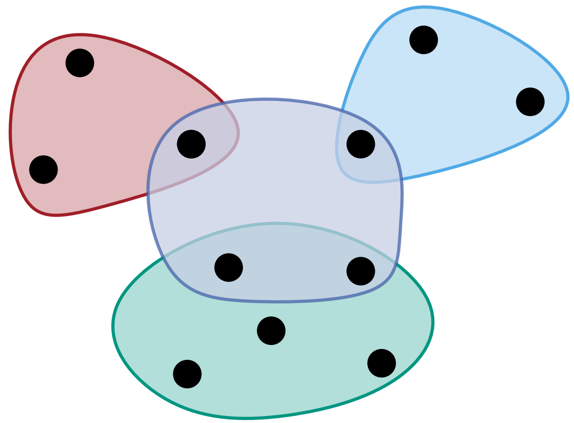
$m = O(n)$



$m = \Theta\left(n^2\right)$

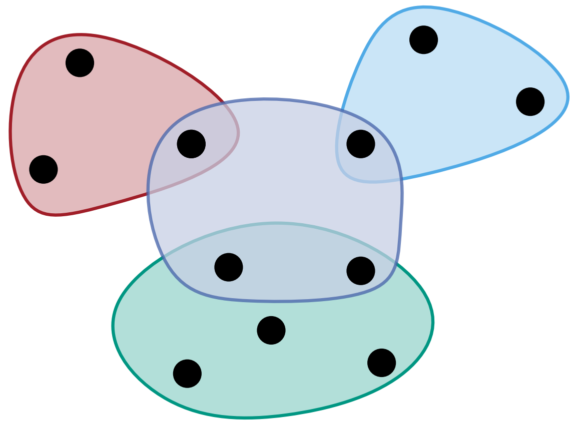
Graphs: Notation & Definitions

- **Hypergraphs**: generalization of graphs
  - hyperedges connect ≥ 2 vertices
  - can represent **d-ary** relationships
  - $E \subseteq \mathcal{P}(V) \setminus \emptyset$

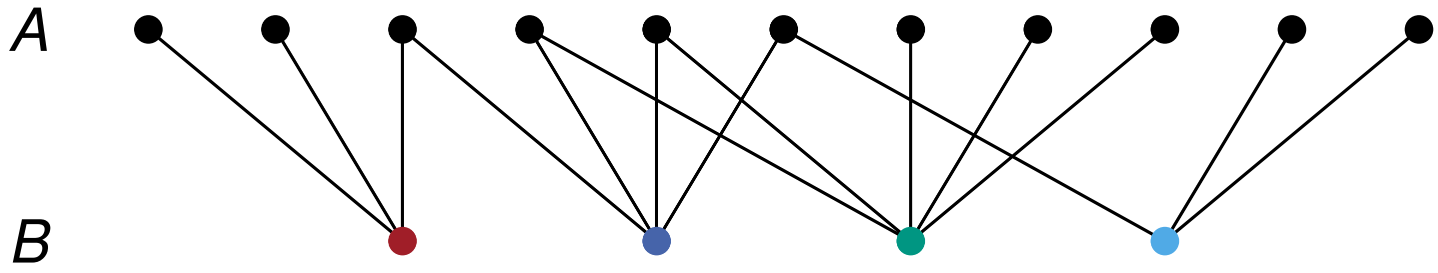


Graphs: Notation & Definitions

- **Hypergraphs**: generalization of graphs
  - hyperedges connect ≥ 2 vertices
  - can represent **d-ary** relationships
  - $E \subseteq \mathcal{P}(V) \setminus \emptyset$

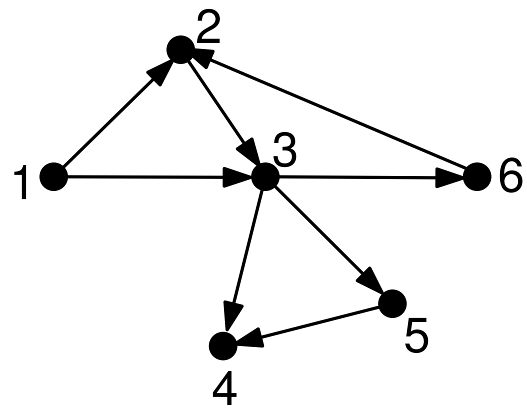


- **Bipartite Graphs**: $\forall (u, v) \in E : (u \in A \land v \in B) \lor (v \in A \land u \in B)$



Graph Representations

- **Unordered Edge Sequence**
  (1,2),(2,3),(4,5),(3,4),(1,3),(3,6),(3,5),(6,2)



+ simple	− navigation in $\Theta(m)$
+ add edges in $O(1)$	− remove edges in $\Theta(m)$

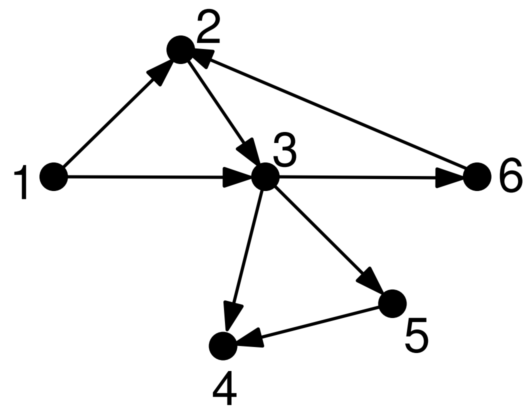
Graph Representations

- **Unordered Edge Sequence**

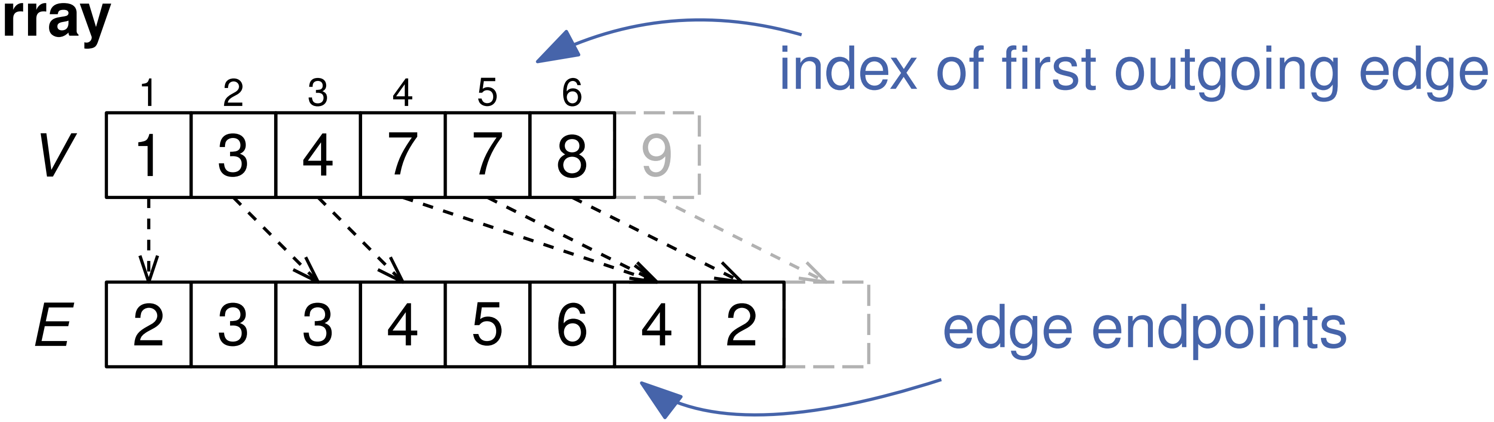
  (1,2),(2,3),(4,5),(3,4),(1,3),(3,6),(3,5),(6,2)

  + simple
  + add edges in $O(1)$

  − navigation in $\Theta(m)$
  − remove edges in $\Theta(m)$



- **Adjacency Array**

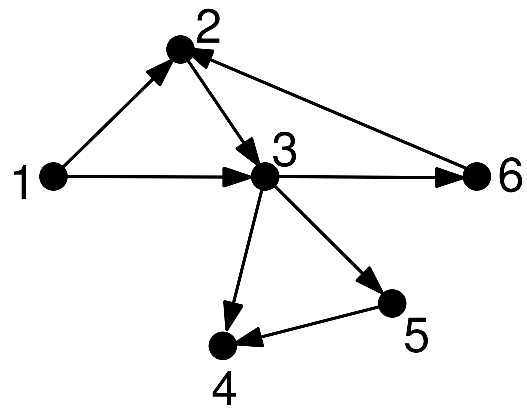


Institute of Theoretical Informatics
Algorithmics Group

Graph Representations

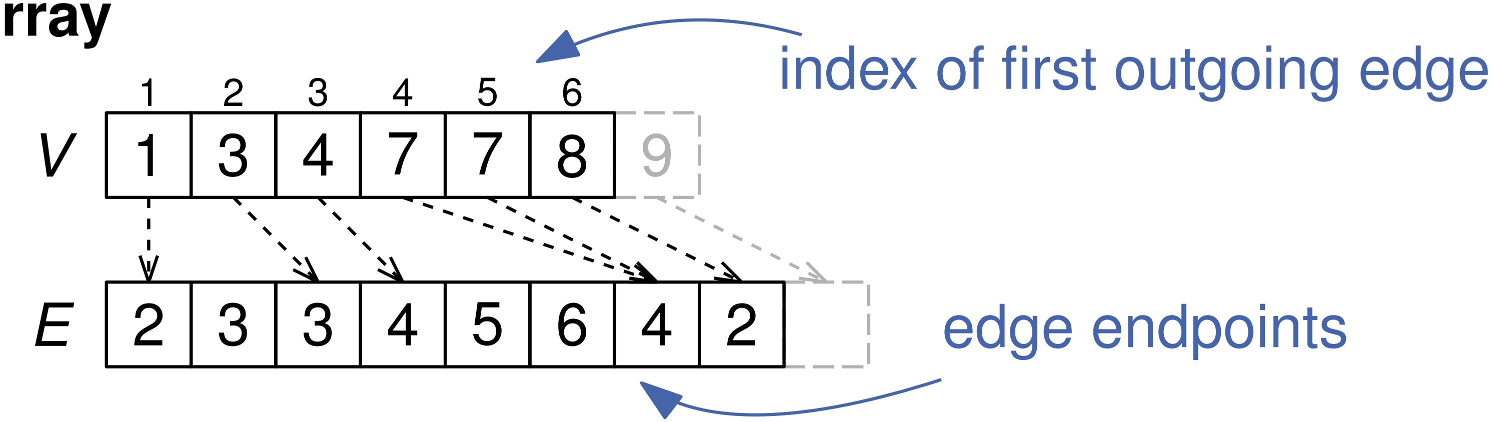
Unordered Edge Sequence

(1,2),(2,3),(4,5),(3,4),(1,3),(3,6),(3,5),(6,2)



- **+** simple
- **+** add edges in $O(1)$
- **−** navigation in $\Theta(m)$
- **−** remove edges in $\Theta(m)$

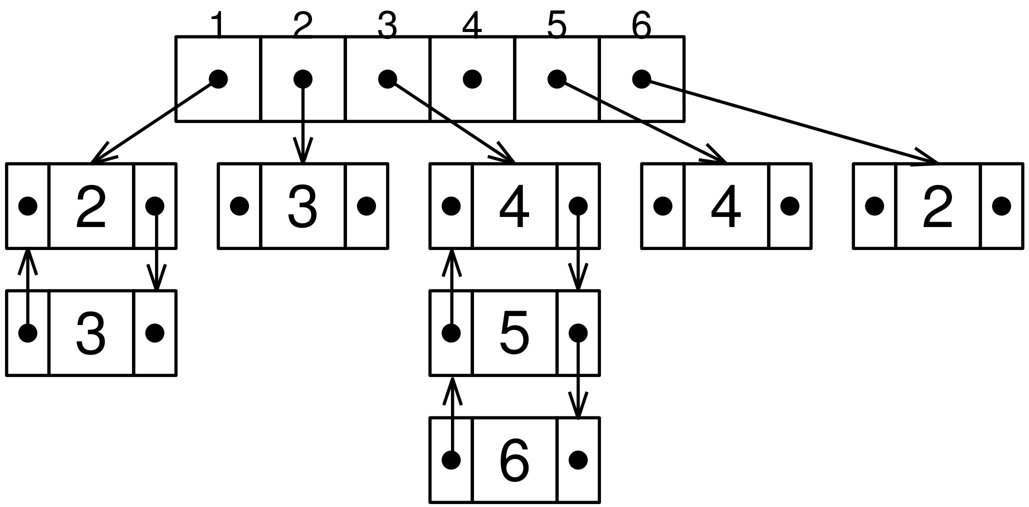
Adjacency Array

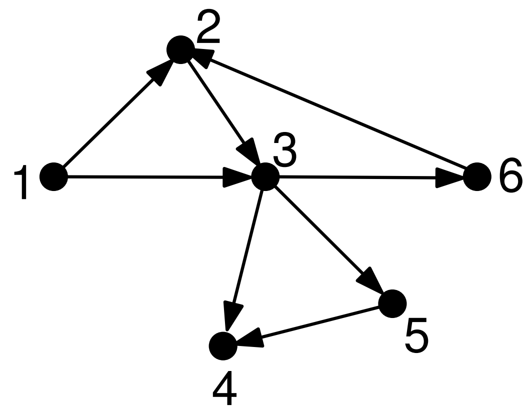


- **+** navigation easy: outgoing edges $E[V[v]], \ldots, E[V[V+1]-1]$
- **+** remove edges: easy via explicit end indices
- **−** add edges

Graph Representations

Adjacency List





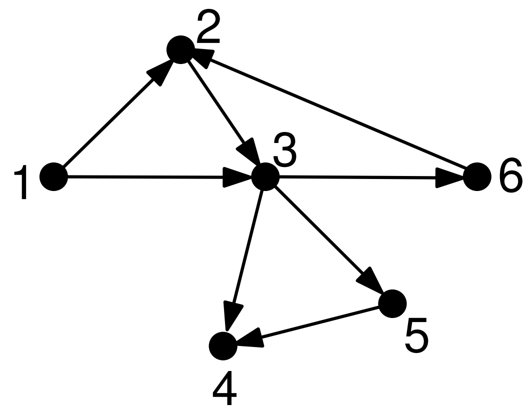
+ adding edges: easy
+ removing edges: easy
+ navigation: easy

– up to 3x more space
– slower (more cache misses)

Graph Representations

Adjacency Matrix

$A \in \{0,1\}^{n \times n}$ with $A(i,j) = [(i,j) \in E]$



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

+ space efficient for very dense graphs
+ query $(u,v) \in E$? easy
+ edge insertions/deletions in $O(1)$
+ connects graph theory with linear algebra

− space inefficient otherwise
− navigation in $O(n)$

Example: $C = A^k \Rightarrow C_{ij} = \#$ paths of length k from i to j

Institute of Theoretical Informatics
Algorithmics Group

Graph Representations

Summary:

- edge sequence
- adjacency array
- adjacency list
- adjacency matrix

no data structure fits all needs!

Graph Representations

Karlsruhe Institute of Technology

Summary:

- edge sequence
- adjacency array
- adjacency list
- adjacency matrix

no data structure fits all needs!

Key Takeaways:

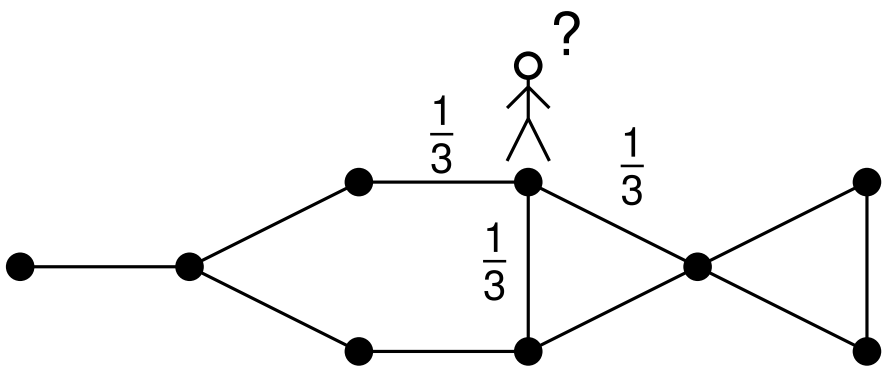
- Choice of DS depends on
  - operations needed
  - frequency of operations
  - static or dynamic?
- Adjacency Array $\rightarrow$ best DS for static graphs
- Matrices rarely used in practice

Graph Traversal

Random Walks

Given undirected Graph $G = (V, E)$

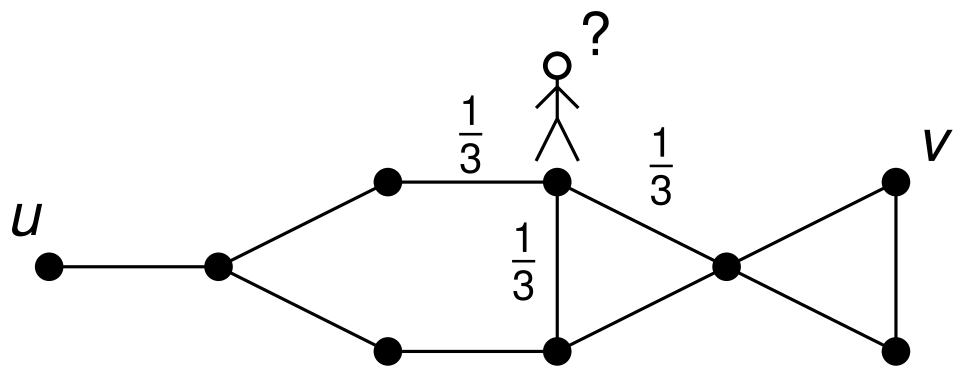
- Random walk in G
  - Random walker that stands at one vertex at each point in time
  - Each edge is taken with same probability



Random Walks

Given undirected Graph $G = (V, E)$

- Random walk in G
  - Random walker that stands at one vertex at each point in time
  - Each edge is taken with same probability



- Interesting properties
  - m_{uv} := expected number of steps from vertex u to v
  - C_{uv} := expected number of steps from vertex u to u via v

Institute of Theoretical Informatics
Algorithmics Group

Applications

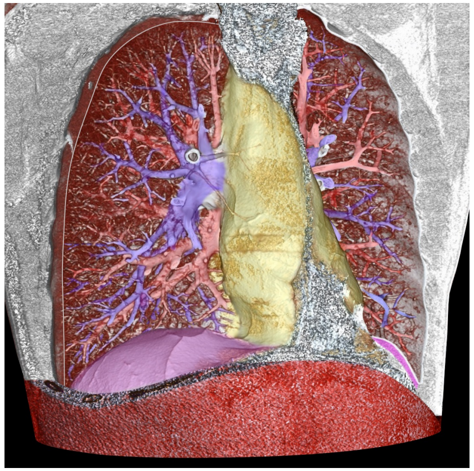
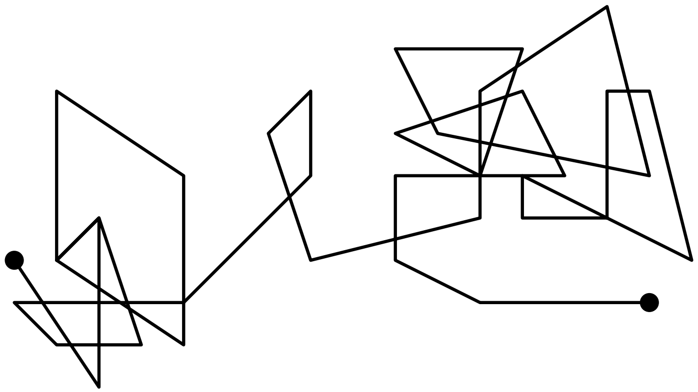
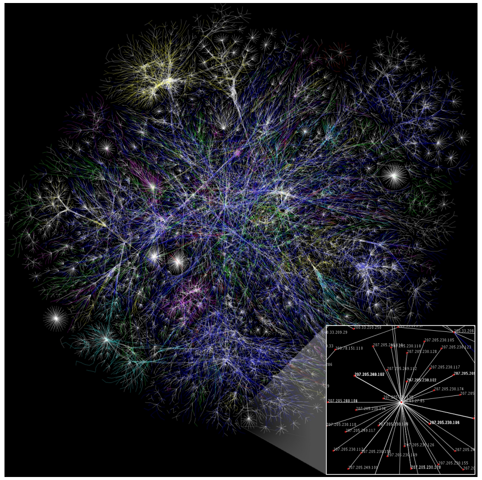


Image segmentation

Model share prices in economics



Model Brownian motion and diffusion

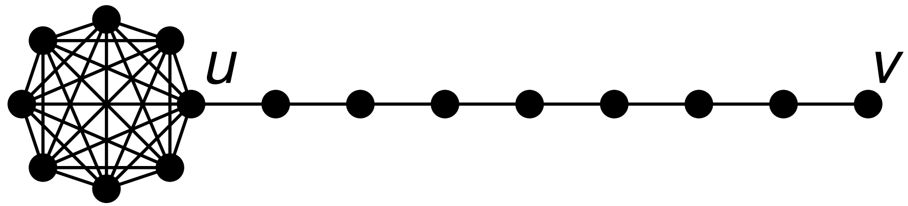


Estimate size of WWW

Example

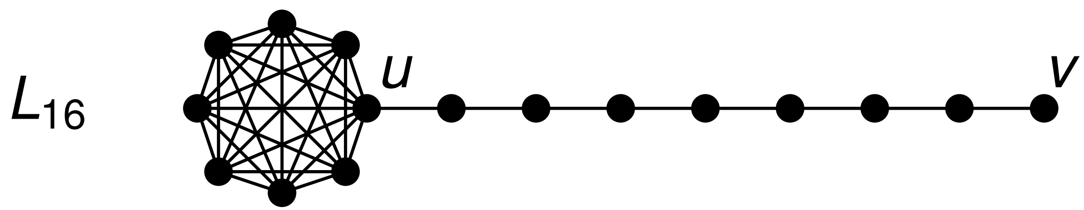
- Lollipop graph L_n
  - First $\frac{n}{2}$ vertices form clique
  - Second $\frac{n}{2}$ vertices form path ”glued” to clique

L_{16}



Example

- Lollipop graph L_n
  - First $\frac{n}{2}$ vertices form clique
  - Second $\frac{n}{2}$ vertices form path ”glued” to clique



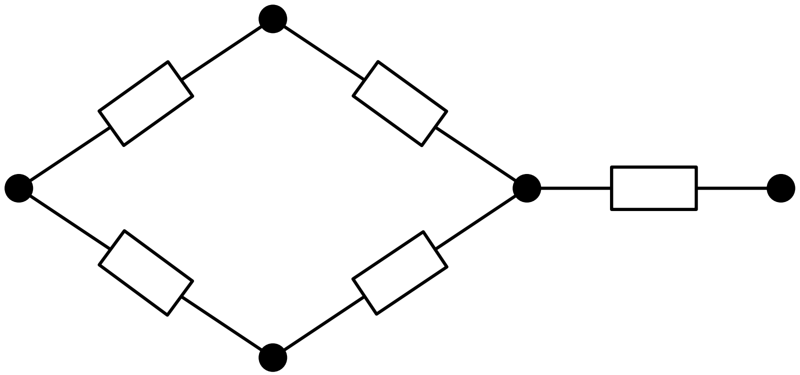
$\Rightarrow m_{uv} \in \Theta(n^3)$

$\Rightarrow m_{vu} \in \Theta(n^2)$

How to efficiently model this problem?

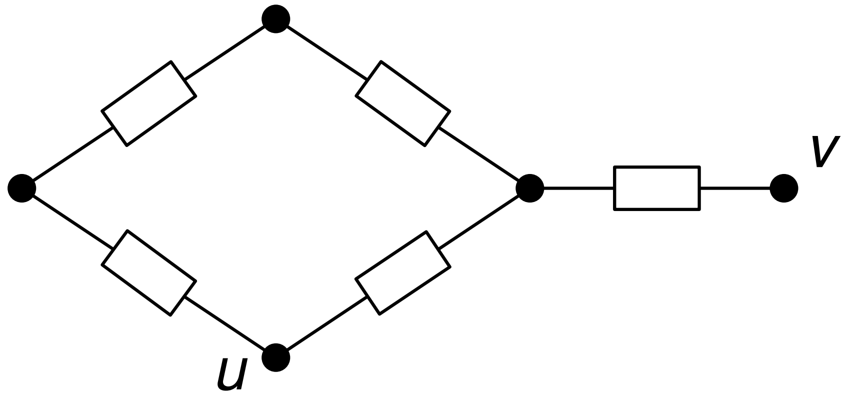
Resistance Networks

- Model graph as network $N(G)$ of electrical resistors
  - Graph has to be undirected, connected and loop-free
  - Replace each edge with resistor of 1Ω



Resistance Networks

- Model graph as network $N(G)$ of electrical resistors
  - Graph has to be undirected, connected and loop-free
  - Replace each edge with resistor of 1Ω

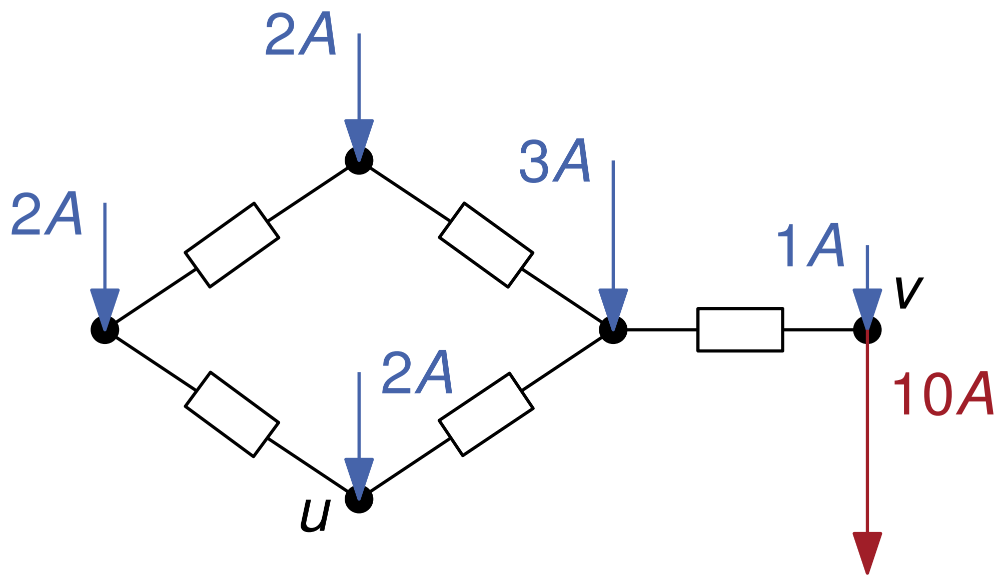


$\Rightarrow$ We can measure the effective resistance R_{uv} between u and v

$\Rightarrow$ We now proof that $C_{uv} = 2mR_{uv}$

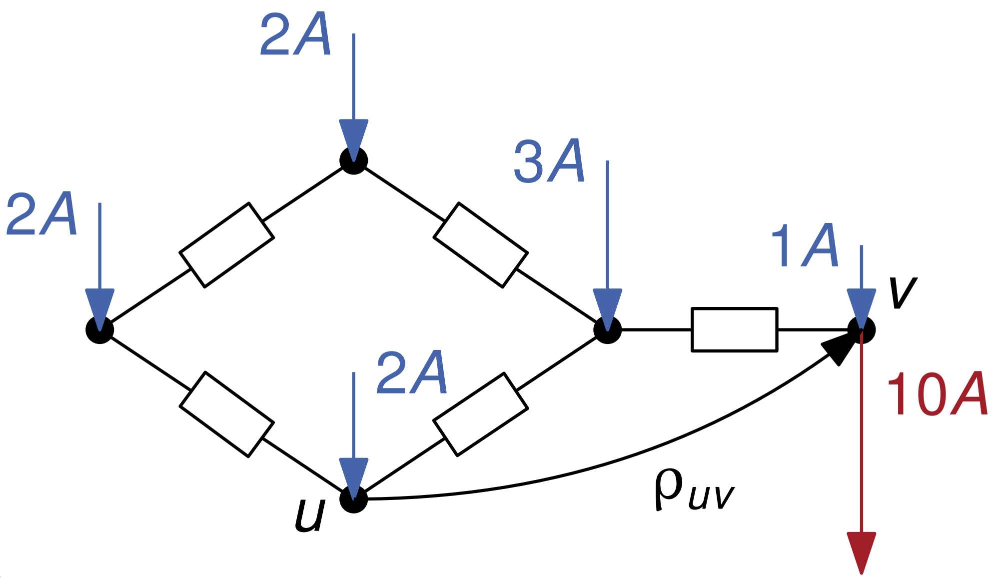
Lemma: $m_{uv} = \rho_{uv}$

- Add electric current $d(x)$ to every vertex $x \in V$
- Remove total current of $2m$ at vertex v



Lemma: $m_{uv} = \rho_{uv}$

- Add electric current $d(x)$ to every vertex $x \in V$
- Remove total current of $2m$ at vertex v



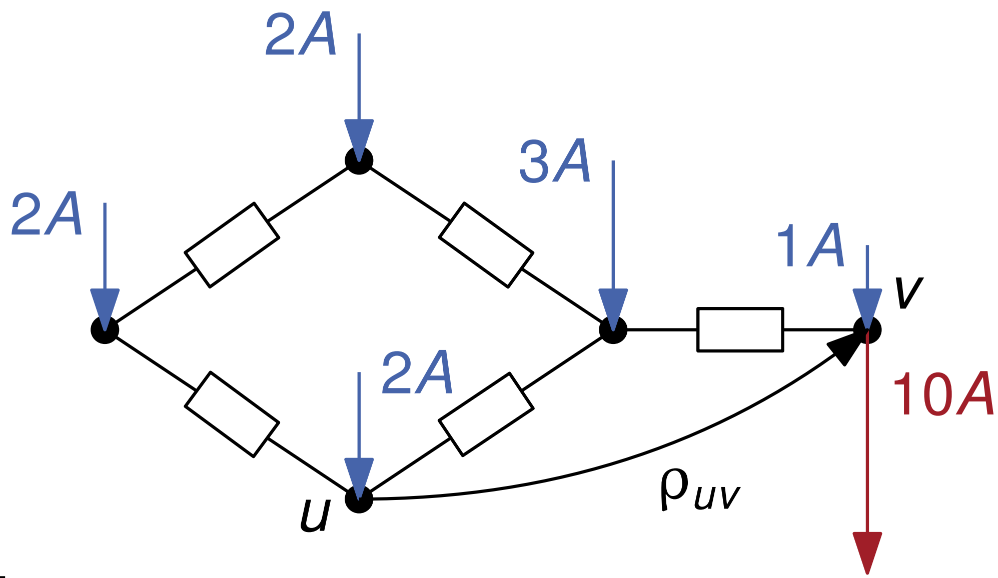
- Kirchoff's law:
  $d(u) = \sum_{w \in \Gamma(u)} (\rho_{uv} - \rho_{wv}) \Leftrightarrow d(u) + \sum_{w \in \Gamma(u)} \rho_{wv} = d(u)\rho_{uv}$

Institute of Theoretical Informatics
Algorithmics Group

Lemma: $m_{uv} = \rho_{uv}$

Karlsruhe Institute of Technology

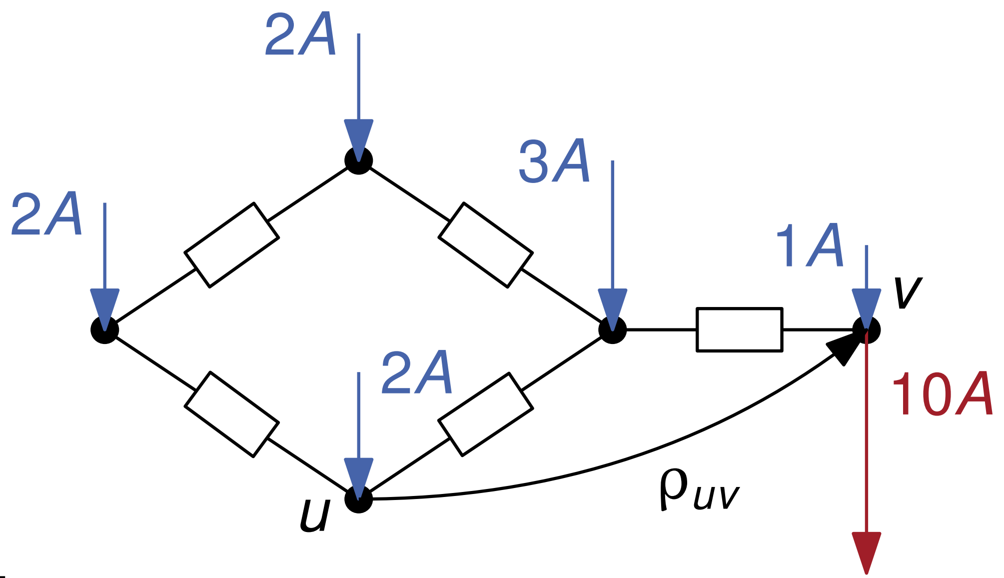
- Add electric current $d(x)$ to every vertex $x \in V$
- Remove total current of $2m$ at vertex v



- Kirchoff's law:
  $d(u) = \sum_{w \in \Gamma(u)} (\rho_{uv} - \rho_{wv}) \Leftrightarrow d(u) + \sum_{w \in \Gamma(u)} \rho_{wv} = d(u)\rho_{uv}$
- Linearity of expectation:
  $m_{uv} = \sum_{w \in \Gamma(u)} (1 + m_{wv})/d(u) \Leftrightarrow d(u) + \sum_{w \in \Gamma(u)} m_{wv} = d(u)m_{uv}$

Lemma: $m_{uv} = \rho_{uv}$

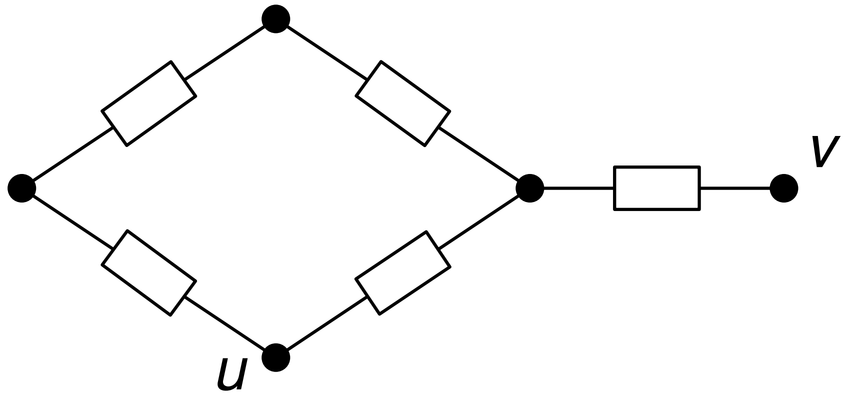
- Add electric current $d(x)$ to every vertex $x \in V$
- Remove total current of $2m$ at vertex v



- Kirchoff's law:
  $d(u) = \sum_{w \in \Gamma(u)} (\rho_{uv} - \rho_{wv}) \Leftrightarrow \boxed{d(u) + \sum_{w \in \Gamma(u)} \rho_{wv} = d(u)\rho_{uv}}$
- Linearity of expectation:
  $m_{uv} = \sum_{w \in \Gamma(u)} (1 + m_{wv})/d(u) \Leftrightarrow \boxed{d(u) + \sum_{w \in \Gamma(u)} m_{wv} = d(u)m_{uv}}$ $\square$

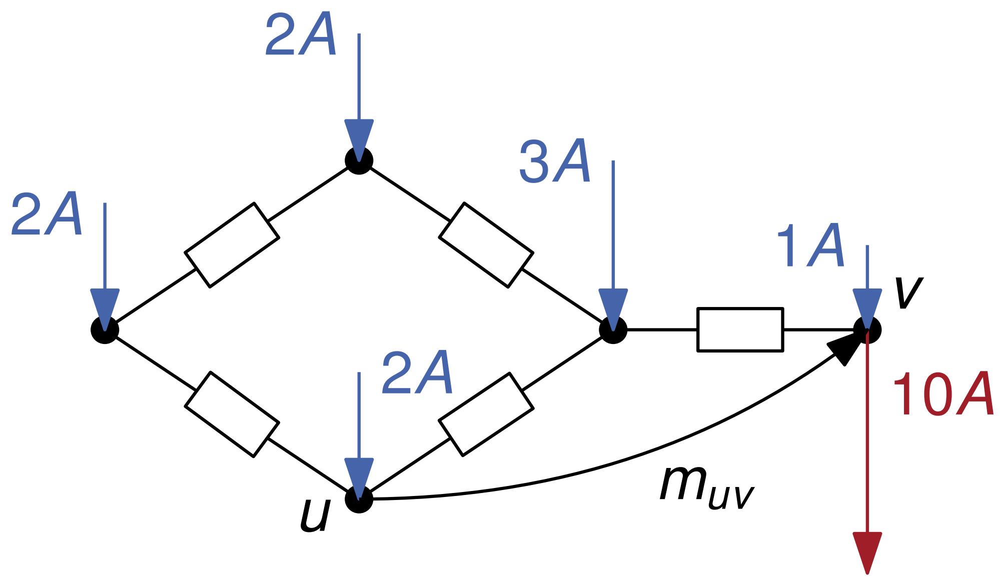
Proof: $C_{uv} = 2mR_{uv}$

- Use $m_{uv} = \rho_{uv}$ and linearity of resistor network
- $C_{uv} = m_{uv} + m_{vu} = \rho_{uv} + \rho_{vu}$



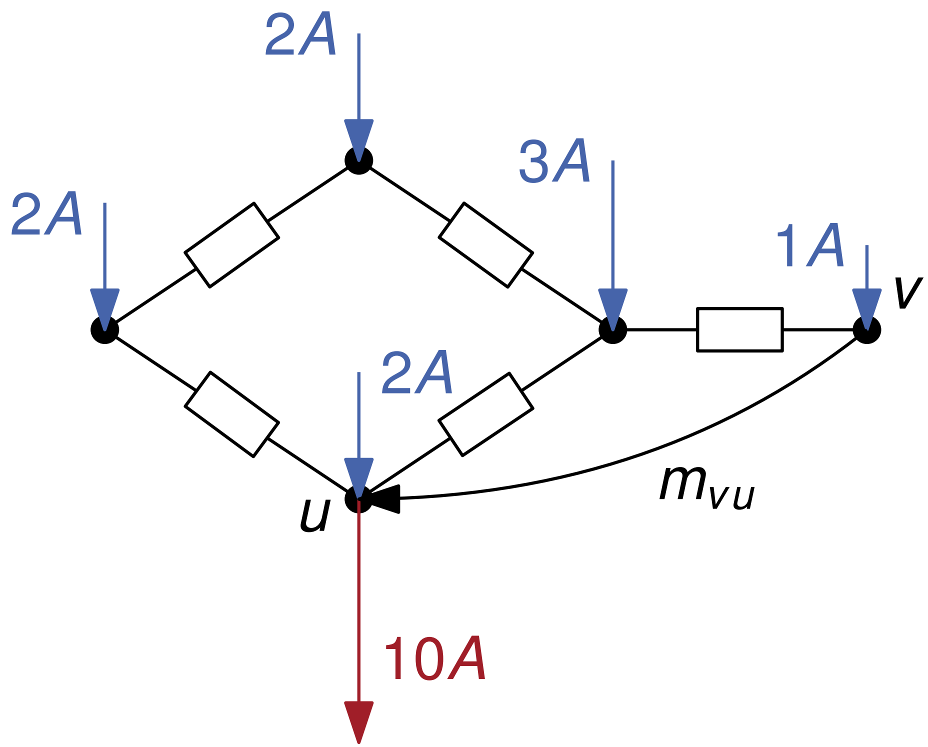
Proof: $C_{uv} = 2mR_{uv}$

- Use $m_{uv} = \rho_{uv}$ and linearity of resistor network
- $C_{uv} = m_{uv} + m_{vu} = \rho_{uv} + \rho_{vu}$



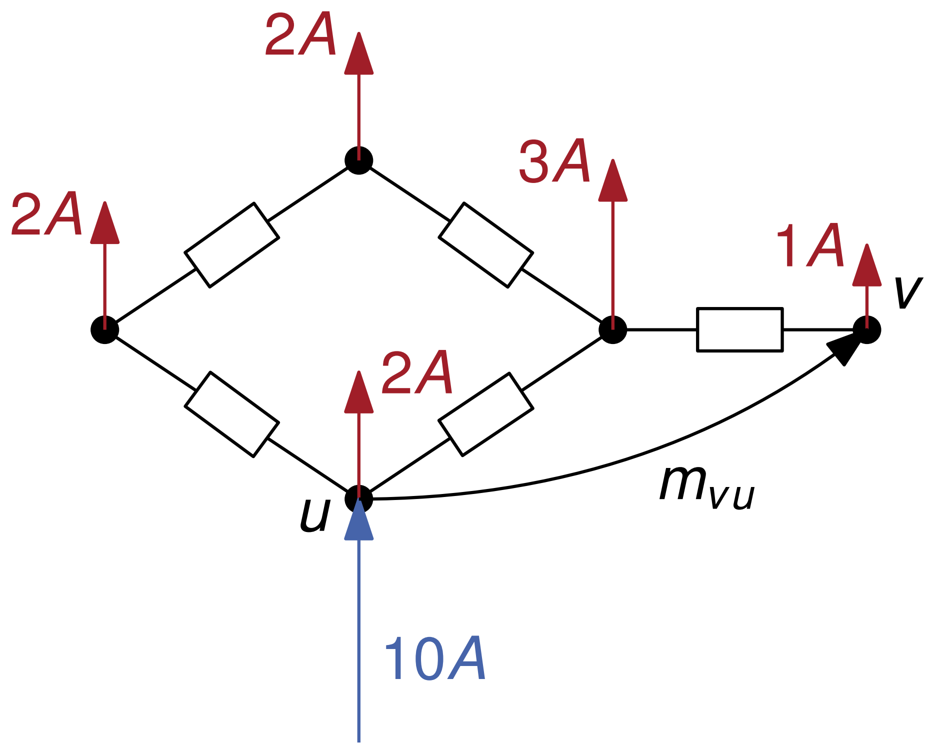
Proof: $C_{uv} = 2mR_{uv}$

- Use $m_{uv} = \rho_{uv}$ and linearity of resistor network
- $C_{uv} = m_{uv} + m_{vu} = \rho_{uv} + \rho_{vu}$



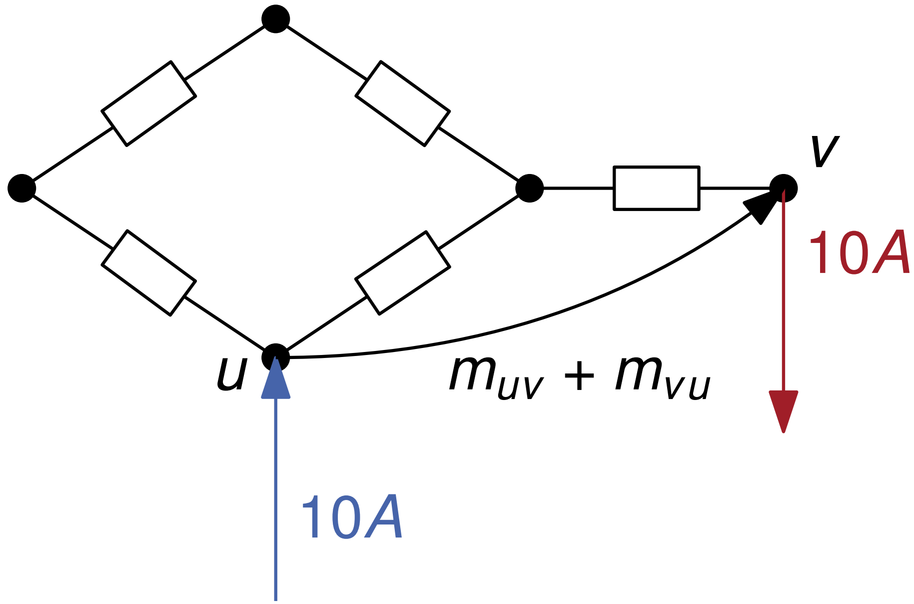
Proof: $C_{uv} = 2mR_{uv}$

- Use $m_{uv} = \rho_{uv}$ and linearity of resistor network
- $C_{uv} = m_{uv} + m_{vu} = \rho_{uv} + \rho_{vu}$



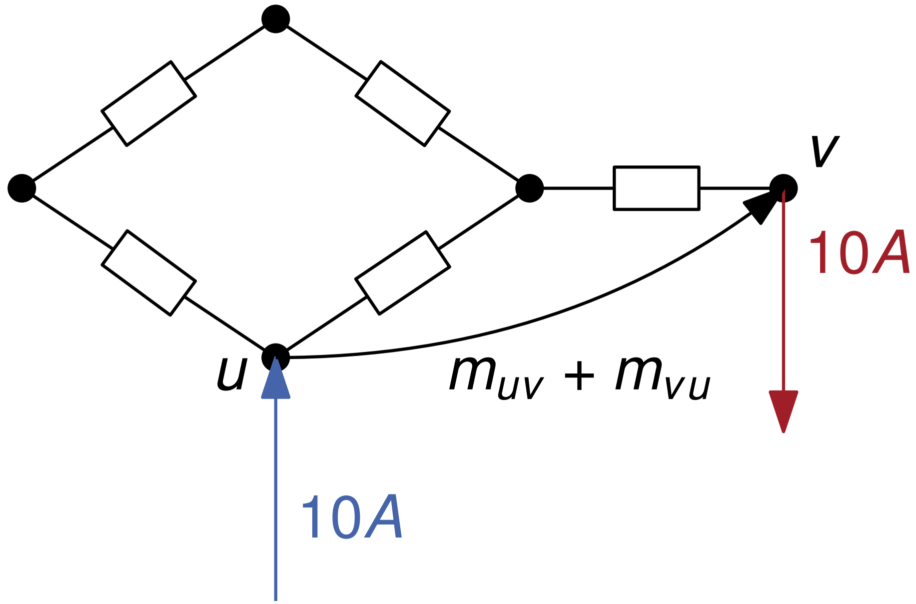
Proof: $C_{uv} = 2mR_{uv}$

- Use $m_{uv} = \rho_{uv}$ and linearity of resistor network
- $C_{uv} = m_{uv} + m_{vu} = \rho_{uv} + \rho_{vu}$



Proof: $C_{uv} = 2mR_{uv}$

- Use $m_{uv} = \rho_{uv}$ and linearity of resistor network
- $C_{uv} = m_{uv} + m_{vu} = \rho_{uv} + \rho_{vu}$



$\Rightarrow$ Ohm's law: $C_{uv} = 2mR_{uv}$

$\square$

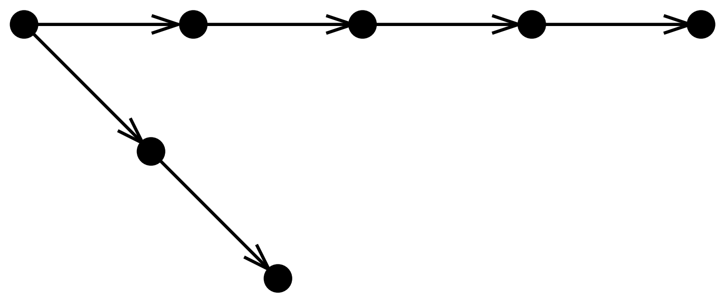
Graph Traversal

Karlsruhe Institute of Technology

Systematic Graph Exploration

- basis of almost all nontrivial graph algorithms
- goal: inspect each edge exactly once
- 2 Algorithms
  - **B**readth-**F**irst **S**earch
  - **D**epth-**F**irst **S**earch

Both construct forests & partition edges into one of 4 classes:

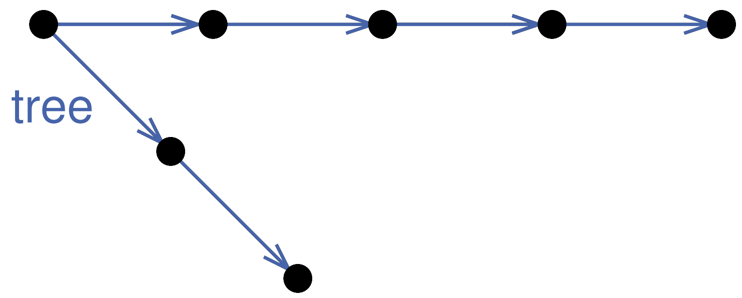


Graph Traversal

Systematic Graph Exploration

- basis of almost all nontrivial graph algorithms
- goal: inspect each edge exactly once
- 2 Algorithms
  - **B**readth-**F**irst **S**earch
  - **D**epth-**F**irst **S**earch

Both construct forests & partition edges into one of 4 classes:

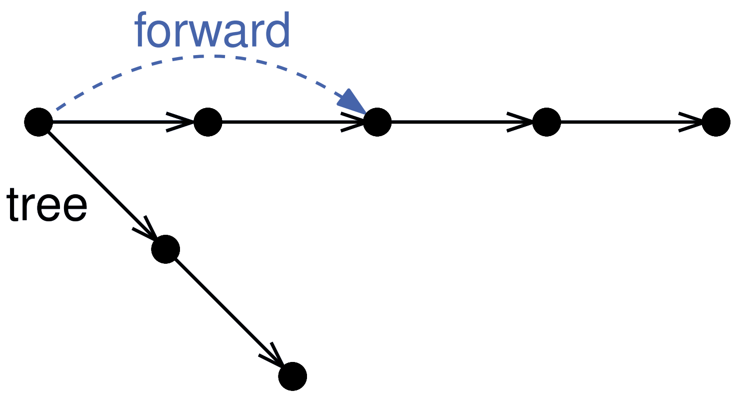


Graph Traversal

Systematic Graph Exploration

- basis of almost all nontrivial graph algorithms
- goal: inspect each edge exactly once
- 2 Algorithms
  - **B**readth-**F**irst **S**earch
  - **D**epth-**F**irst **S**earch

Both construct forests & partition edges into one of 4 classes:

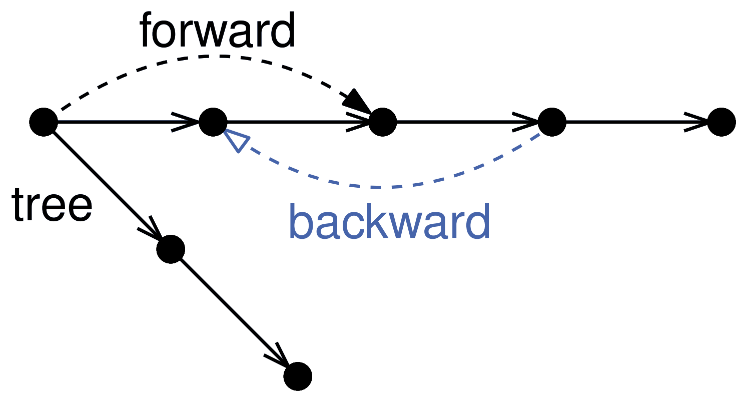


Graph Traversal

Systematic Graph Exploration

- basis of almost all nontrivial graph algorithms
- goal: inspect each edge exactly once
- 2 Algorithms
  - **B**readth-**F**irst **S**earch
  - **D**epth-**F**irst **S**earch

Both construct forests & partition edges into one of 4 classes:



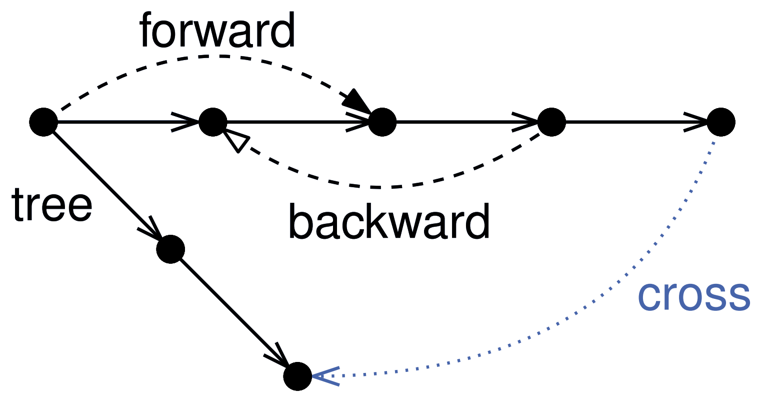
Institute of Theoretical Informatics
Algorithmics Group

Graph Traversal

Systematic Graph Exploration

- basis of almost all nontrivial graph algorithms
- goal: inspect each edge exactly once
- 2 Algorithms
  - **B**readth-**F**irst **S**earch
  - **D**epth-**F**irst **S**earch

Both construct forests & partition edges into one of 4 classes:



Breadth First Search

Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

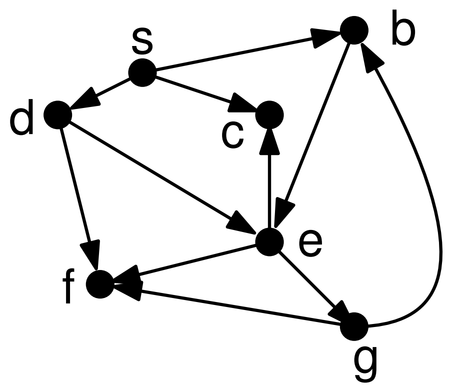
Institute of Theoretical Informatics
Algorithmics Group

Breadth First Search

Build tree starting from **root node** *s* that connects all nodes reachable from *s* via **shortest** paths.

Function bfs(s) :
  $Q{:=}\ \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q{:=}\ Q'$

Graph

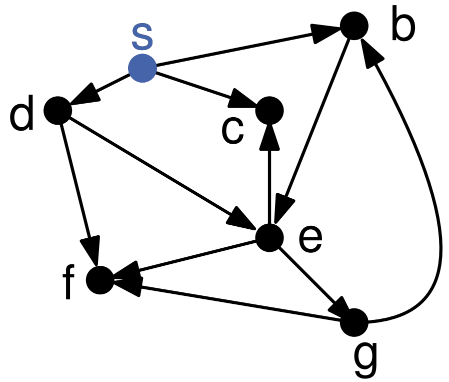


Breadth First Search

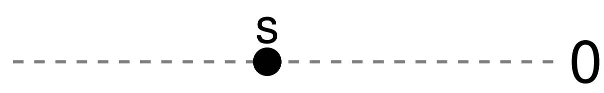
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree



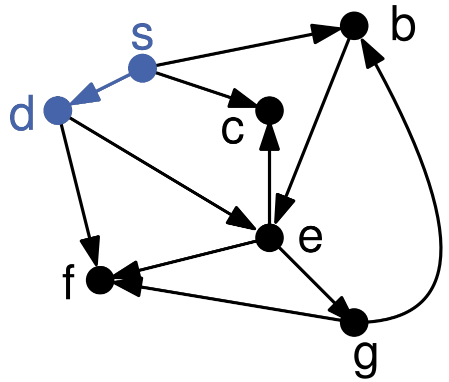
Institute of Theoretical Informatics
Algorithmics Group

Breadth First Search

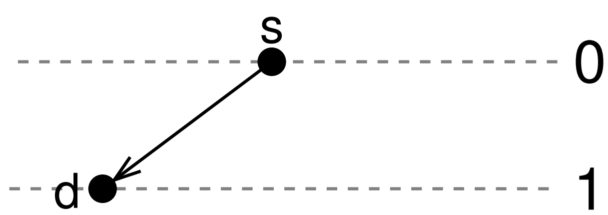
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree

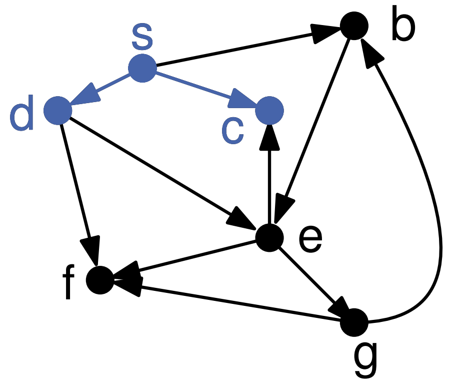


Breadth First Search

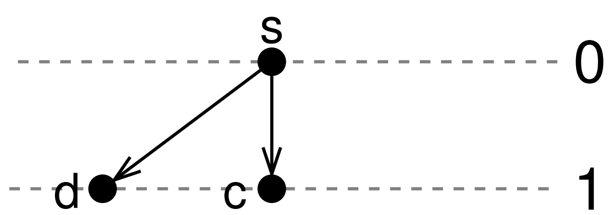
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree

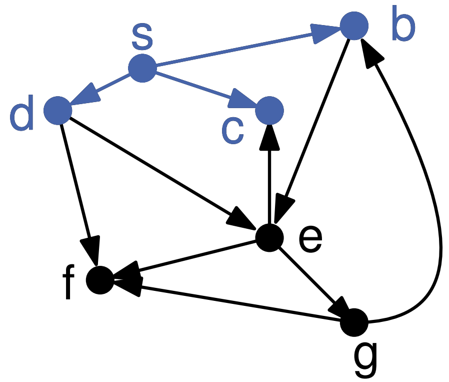


Breadth First Search

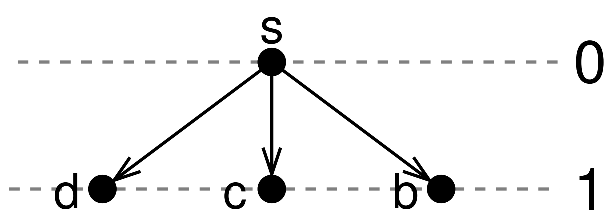
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function $\mathrm{bfs}(s)$:
- $Q := \langle s \rangle$ // current layer
- **while** $Q \neq \langle\rangle$ **do**
  - explore nodes Q
  - remember node in next layer in Q'
  - $Q := Q'$

Graph



BFS-Tree



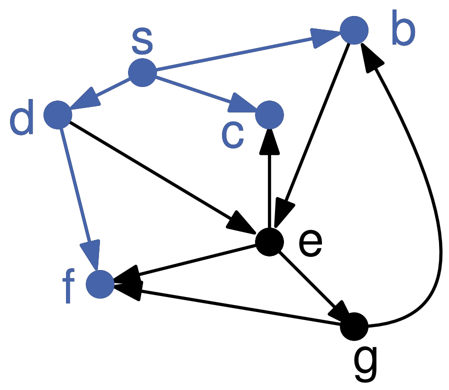
Institute of Theoretical Informatics
Algorithmics Group

Breadth First Search

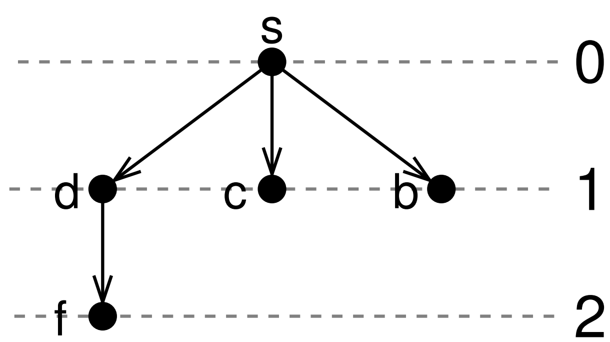
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function $\text{bfs}(s)$:
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree



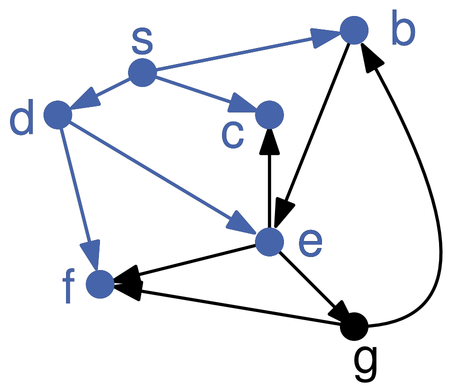
Institute of Theoretical Informatics
Algorithmics Group

Breadth First Search

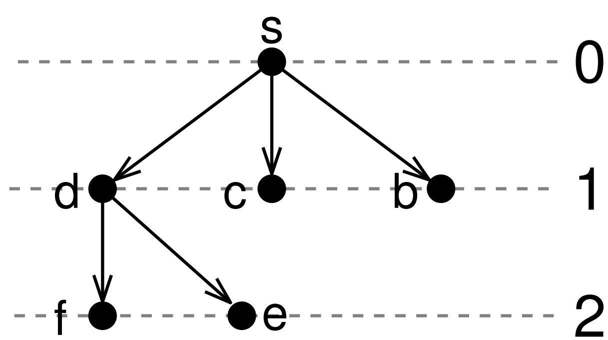
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree

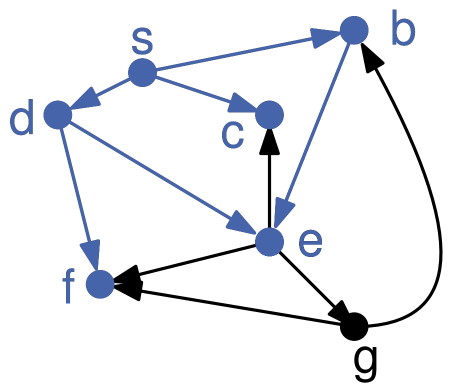


Breadth First Search

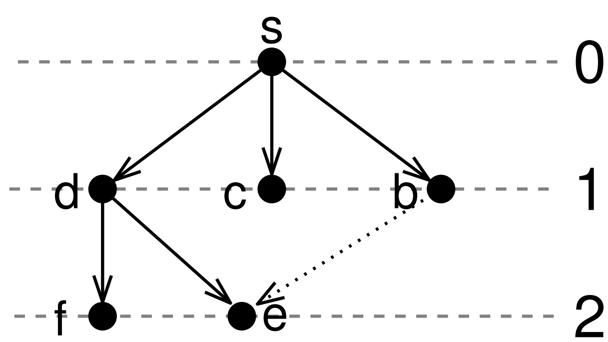
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree

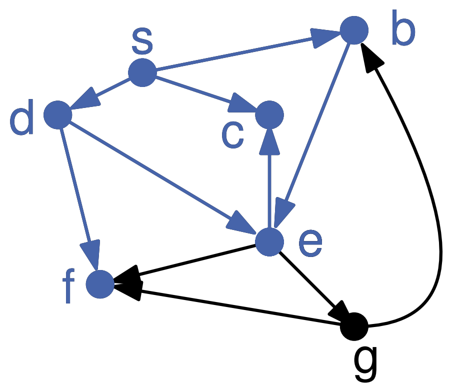


Breadth First Search

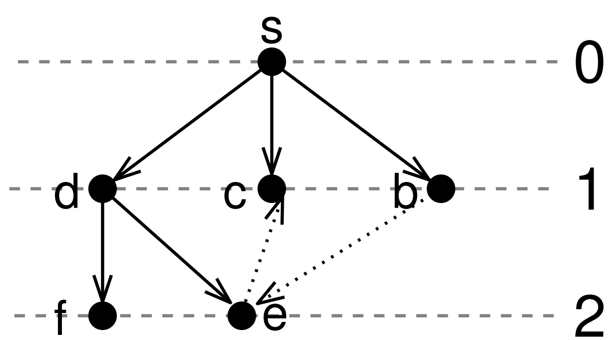
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

Build tree starting from **root node** *s* that connects all nodes reachable from *s* via **shortest** paths.

Function bfs(s) :

$Q := \langle s \rangle$ // current layer

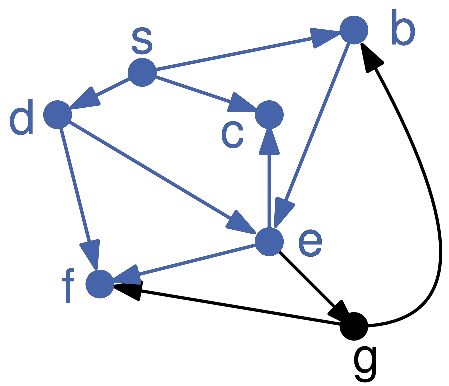
while $Q \neq \langle\rangle$ **do**

explore nodes Q

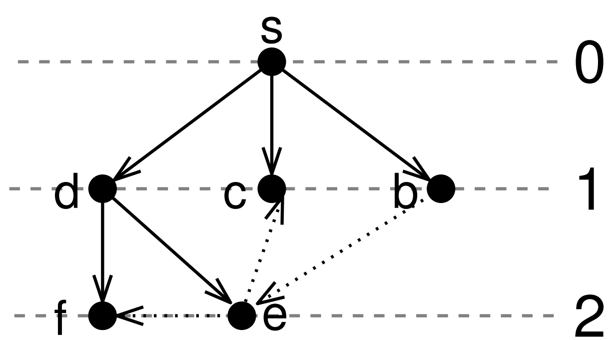
remember node in next layer in Q'

$Q := Q'$

Graph



BFS-Tree

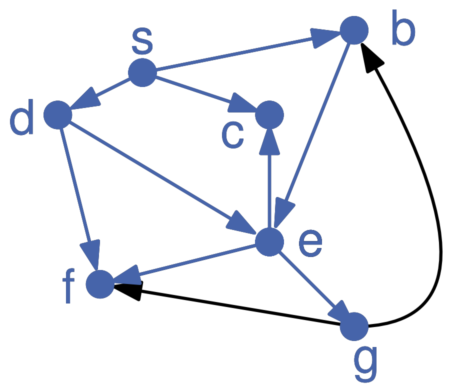


Breadth First Search

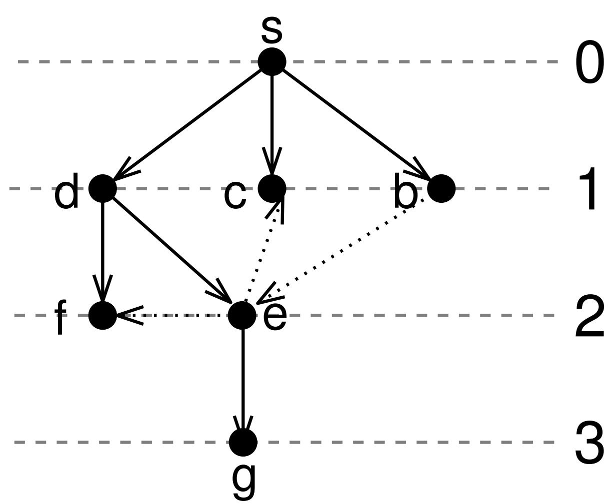
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
$Q := \langle s \rangle$ // current layer
while $Q \neq \langle\rangle$ **do**
explore nodes Q
remember node in next layer in Q'
$Q := Q'$

Graph



BFS-Tree

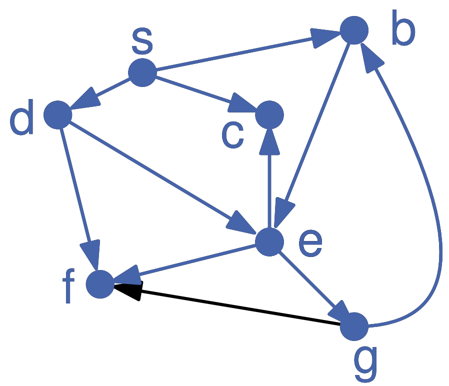


Breadth First Search

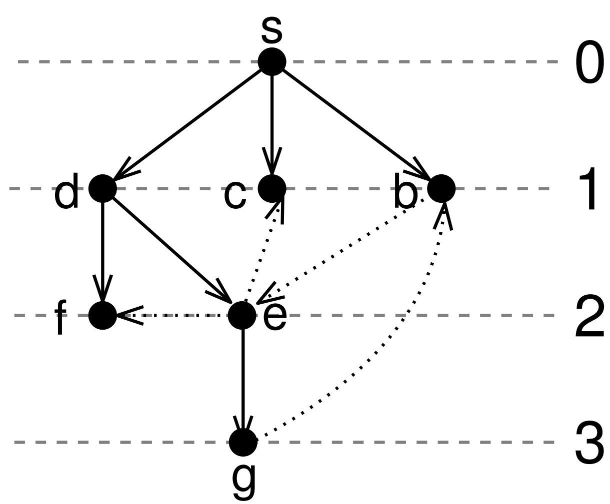
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree

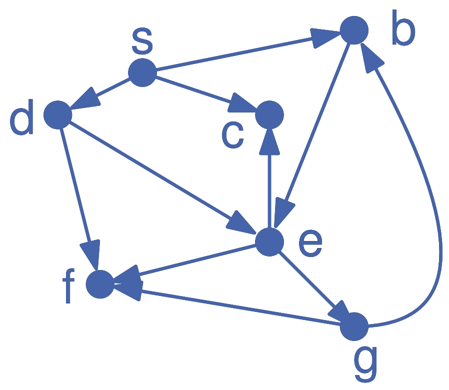


Breadth First Search

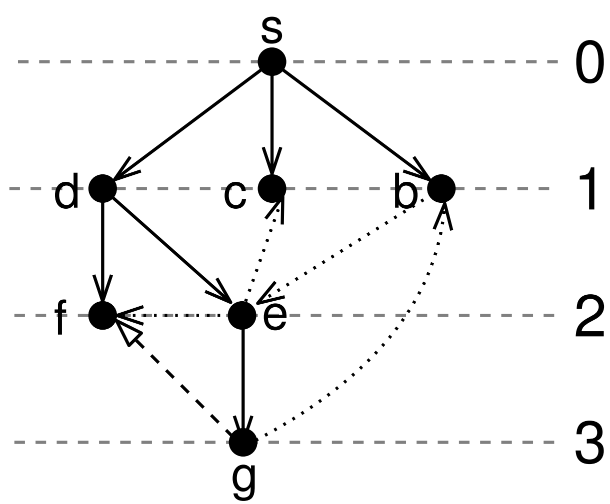
Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
  $Q := \langle s \rangle$ // current layer
  while $Q \neq \langle\rangle$ **do**
    explore nodes Q
    remember node in next layer in Q'
    $Q := Q'$

Graph



BFS-Tree

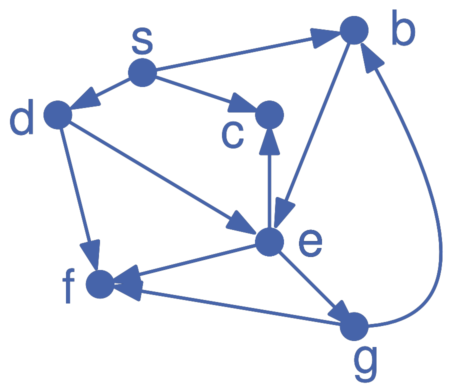


Breadth First Search

Build tree starting from **root node** s that connects all nodes reachable from s via **shortest** paths.

Function bfs(s) :
    $Q := \langle s \rangle$ // current layer
    while $Q \neq \langle\rangle$ **do**
        explore nodes Q
        remember node in next layer in Q'
        $Q := Q'$

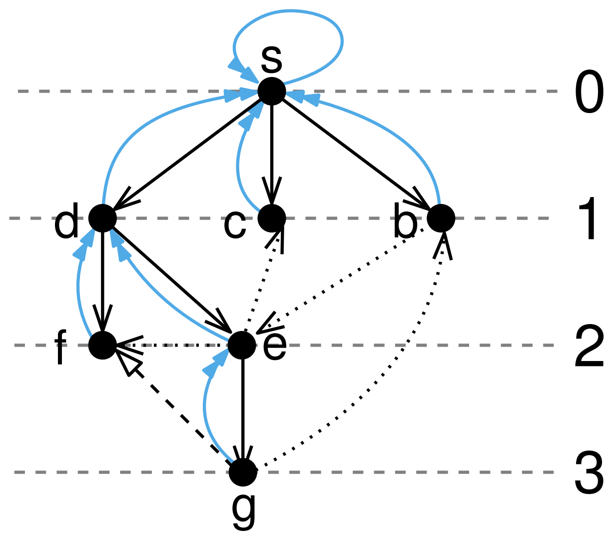
Graph



How to store the tree?

- array stores parents
- not reached: parent$[v] = \perp$
- root: parent$[s] = s$

BFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

Depth First Search

Explore the graph as far as possible along each branch and return only if you run out of options.

init
foreach $s \in V$ **do**
  if s is not marked **then**
    mark s // make s a root and grow
    root(s) // a new DFS tree rooted at s
    DFS(s, s)

init:
  dfsPos=1 : 1..n
  finishingTime=1 : 1..n

root(s):
  dfsNum[s]:= dfsPos++

Institute of Theoretical Informatics
Algorithmics Group

Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

Institute of Theoretical Informatics
Algorithmics Group

Depth First Search

Karlsruhe Institute of Technology

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Institute of Theoretical Informatics
Algorithmics Group

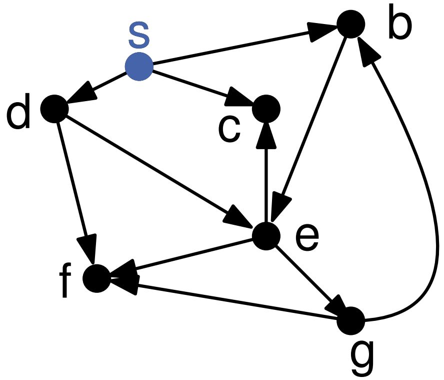
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree

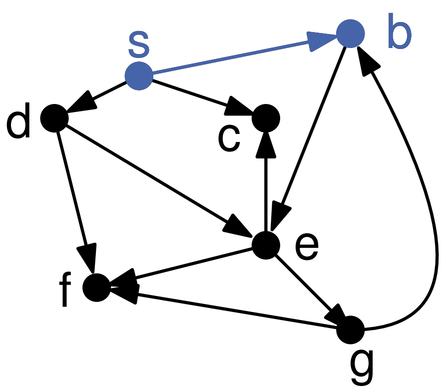
Depth First Search

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

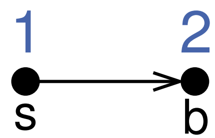
traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

Depth First Search

Procedure DFS(u, v : NodeId)

    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

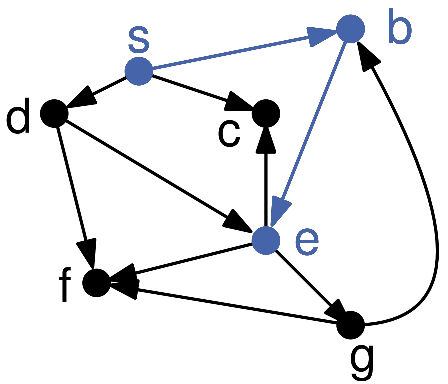
traverseTreeEdge(v, w):

    dfsNum[w]:= dfsPos++

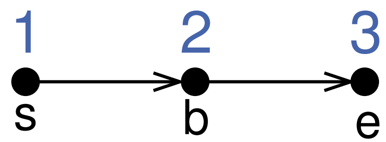
backtrack(u, v):

    finishTime[v]:= finishingTime++

Graph



DFS-Tree



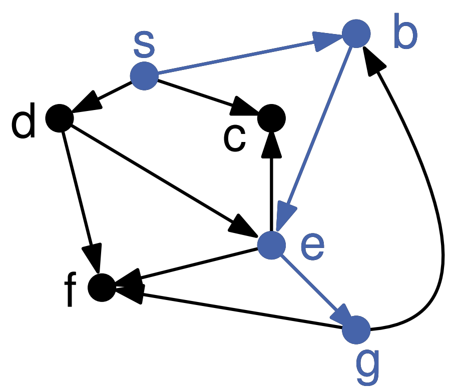
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

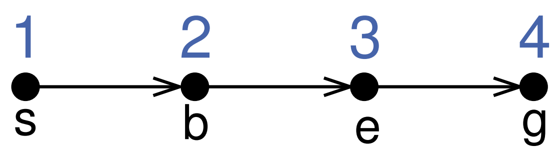
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



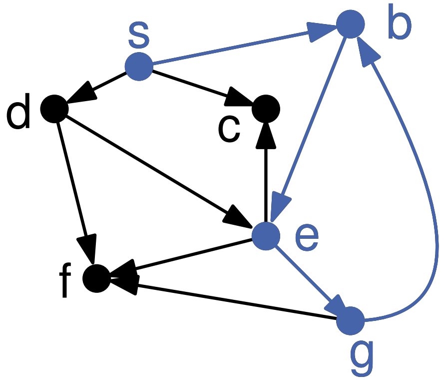
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

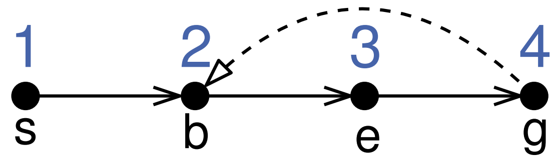
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



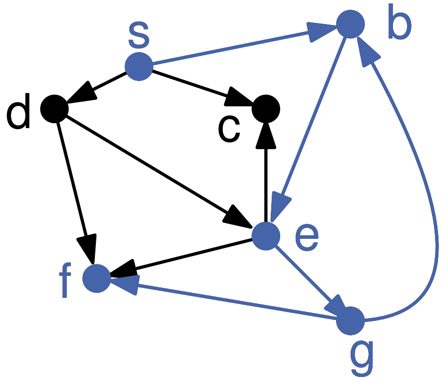
Depth First Search

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

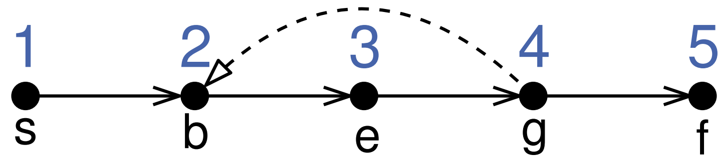
traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Graph



DFS-Tree



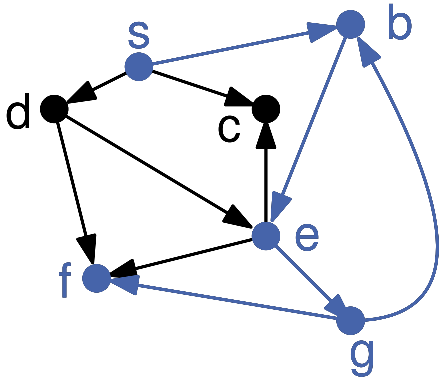
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

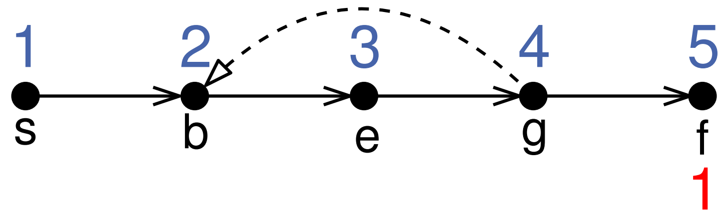
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

Depth First Search

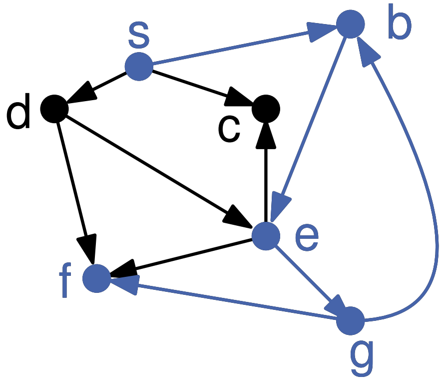
Karlsruhe Institute of Technology

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

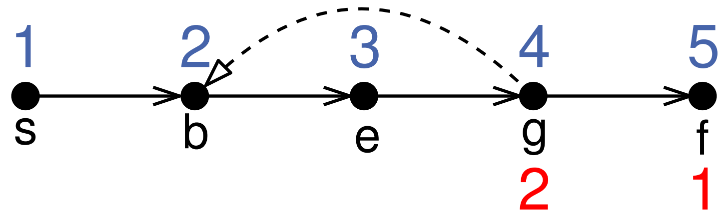
traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Graph



DFS-Tree



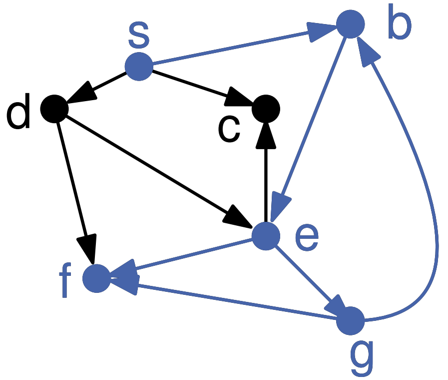
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

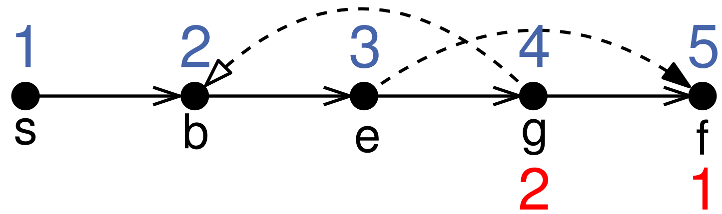
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



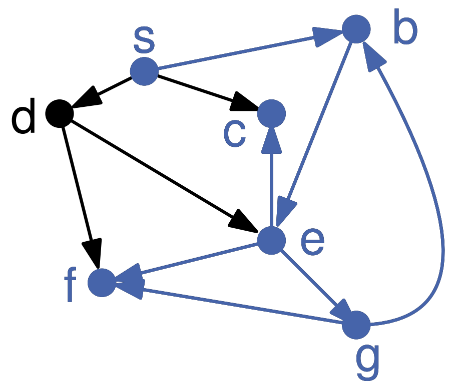
Depth First Search

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

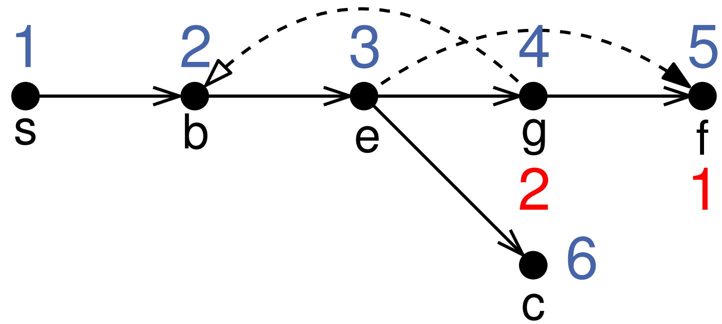
traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

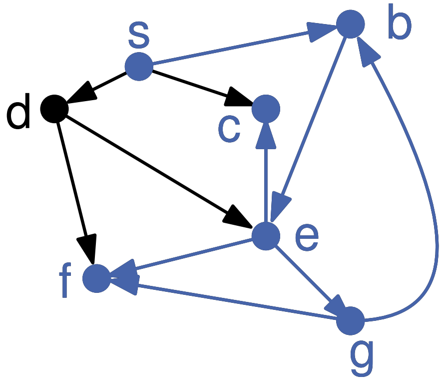
Depth First Search

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

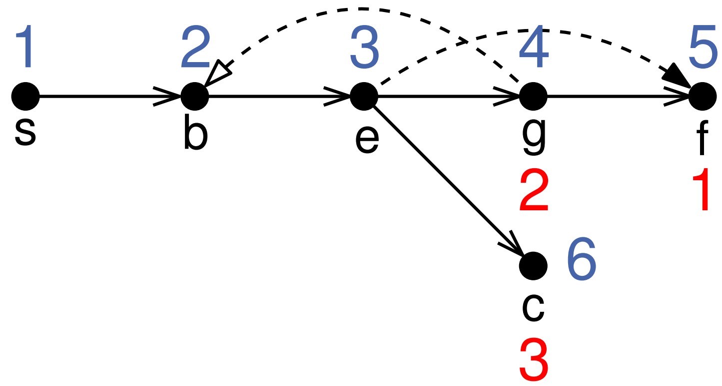
traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

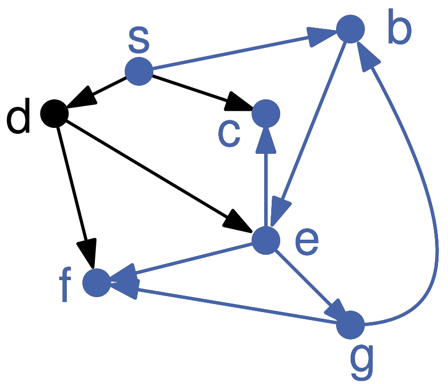
Depth First Search

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

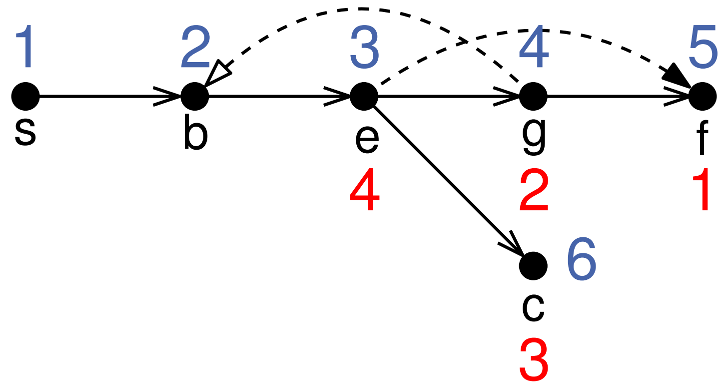
traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

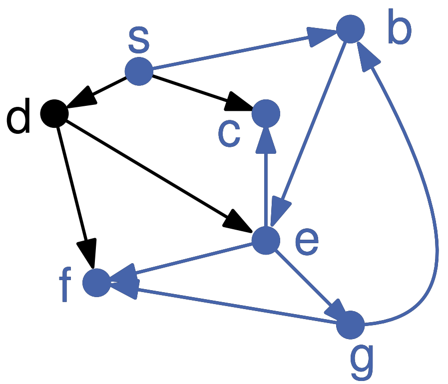
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

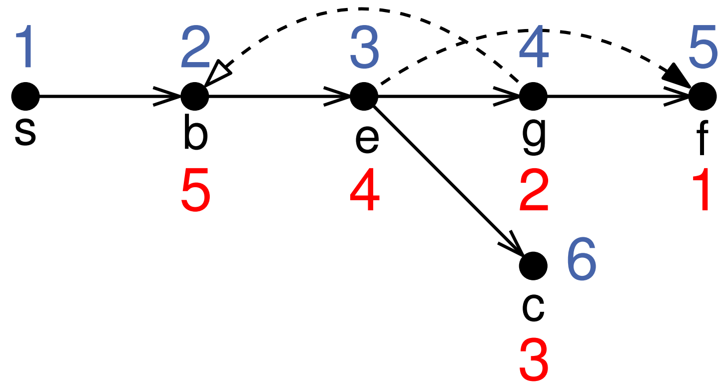
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

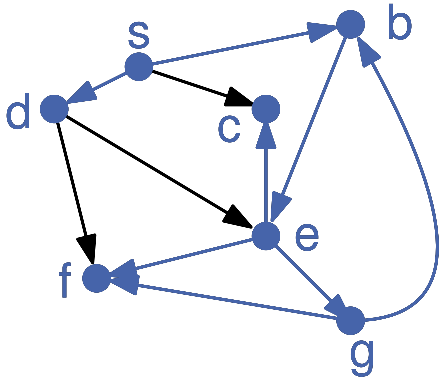
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

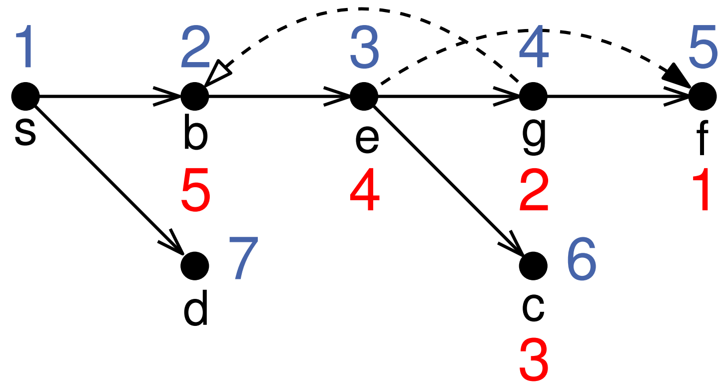
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

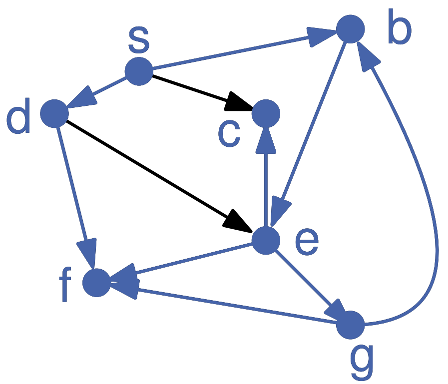
Depth First Search

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

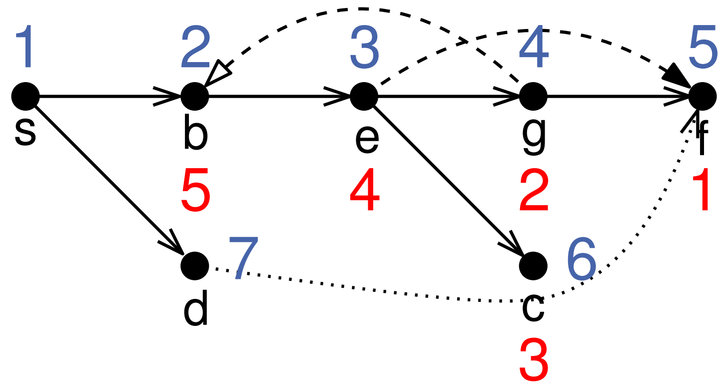
traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Graph



DFS-Tree



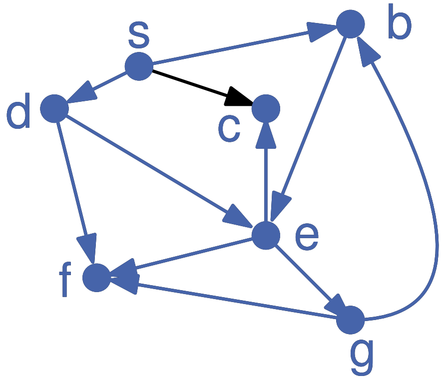
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

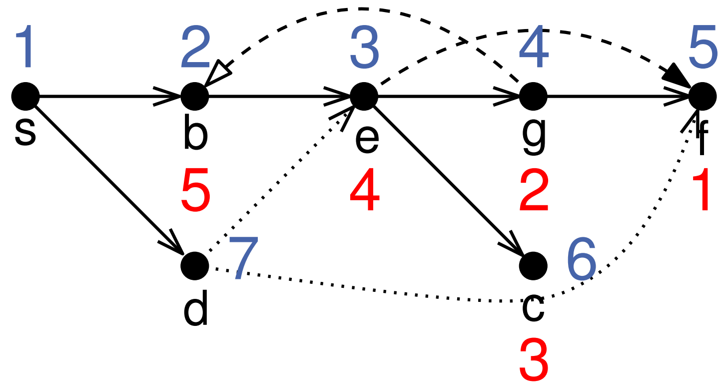
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

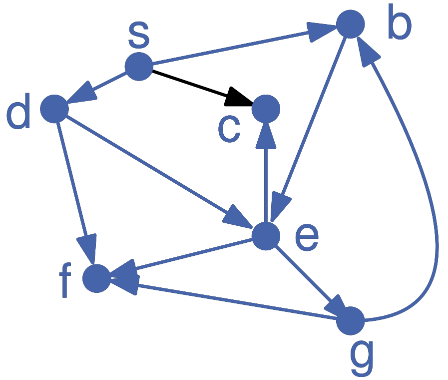
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

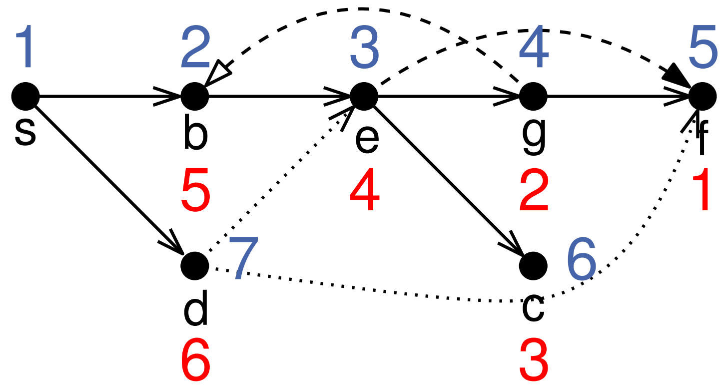
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

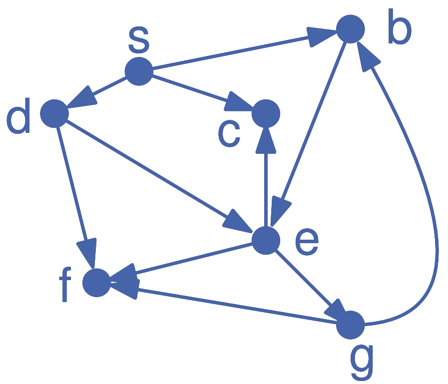
Depth First Search

Procedure DFS(u, v : NodeId)
  foreach $(v, w) \in E$ **do**
    if w is marked **then**
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)

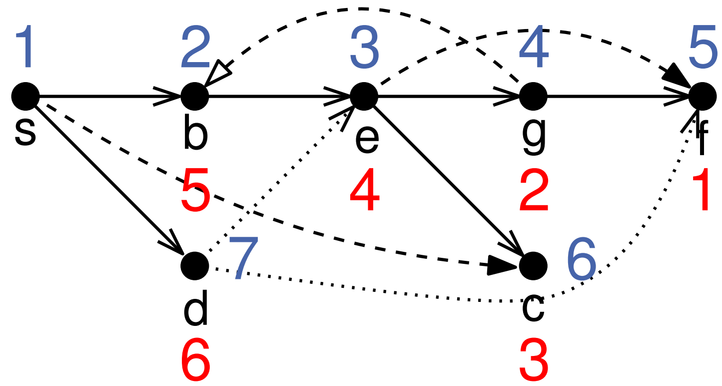
traverseTreeEdge(v, w):
  dfsNum[w]:= dfsPos++

backtrack(u, v):
  finishTime[v]:= finishingTime++

Graph



DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

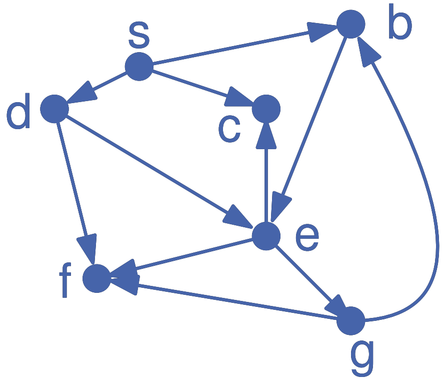
Depth First Search

Procedure DFS(u, v : NodeId)
    foreach $(v, w) \in E$ **do**
        if w is marked **then**
            traverseNonTreeEdge(v, w)
        else
            traverseTreeEdge(v, w)
            mark w
            DFS(v, w)
    backtrack(u, v)

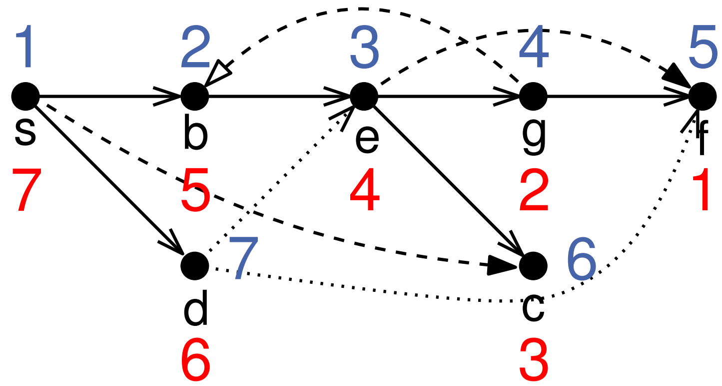
traverseTreeEdge(v, w):
    dfsNum[w]:= dfsPos++

backtrack(u, v):
    finishTime[v]:= finishingTime++

Graph



DFS-Tree

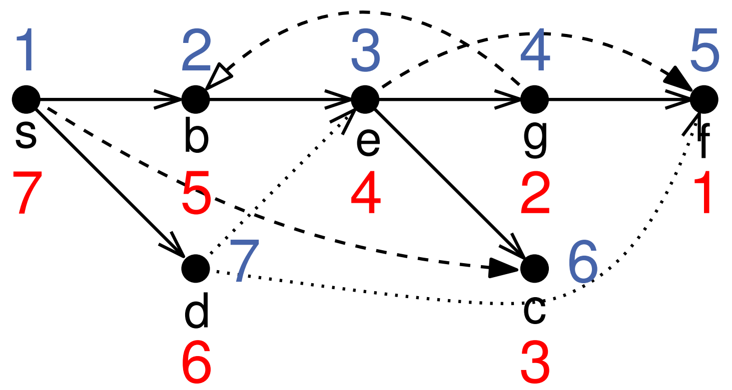


Institute of Theoretical Informatics
Algorithmics Group

DFS: Edge Classification

type (v, w)	$\text{dfsNum}[v] < \text{dfsNum}[w]$	$\text{finishTime}[w] < \text{finishTime}[v]$	w is marked
tree	yes	yes	no
forward	yes	yes	yes
backward	no	no	yes
cross	no	yes	yes

DFS-Tree

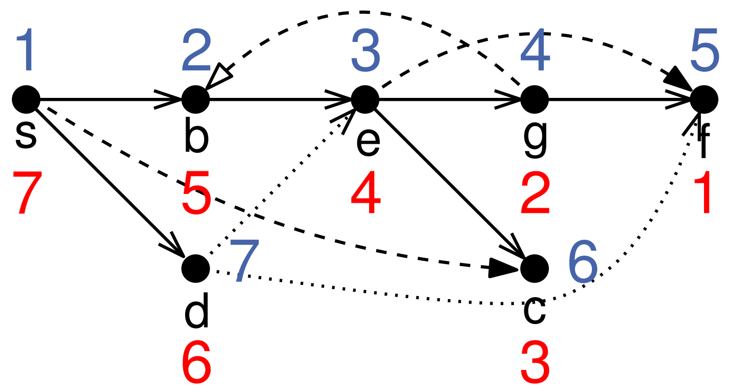


DFS: Edge Classification

Lemma:
The following properties are equivalent:
(i) G is an acyclic directed graph (DAG)
(ii) DFS on G produces no backward edges
(iii) All edges of G go from larger to smaller finishing times

$\Rightarrow$ **Cycle Detection**
$\Rightarrow$ **Topological Sorting**

DFS-Tree



Institute of Theoretical Informatics
Algorithmics Group

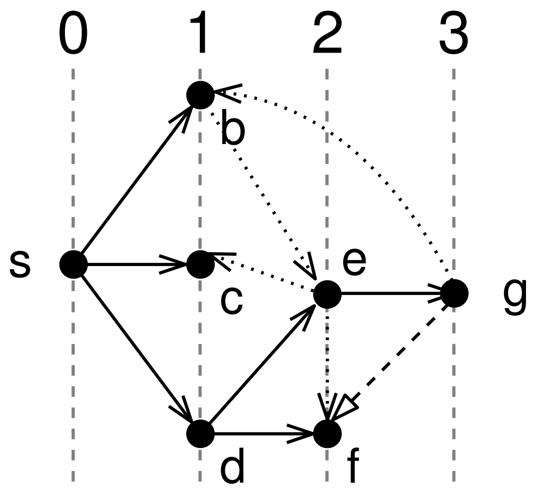
Graph Problems

Institute of Theoretical Informatics
Algorithmics Group

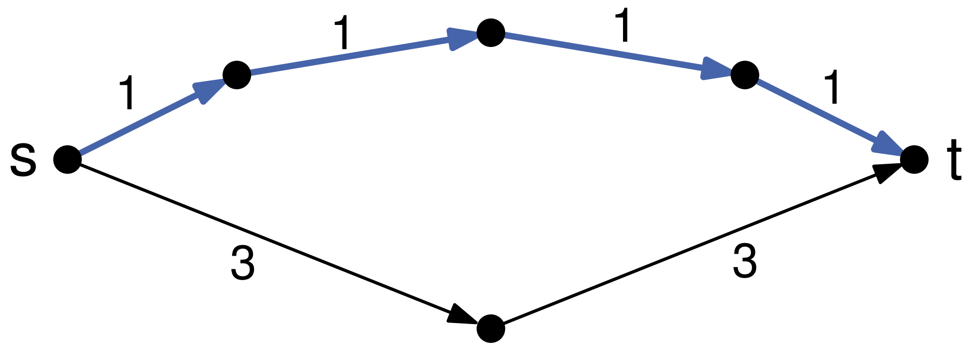
Finding Shortest Paths in Graphs

Unweighted Graphs ($\forall e \in E : \omega(e) = 1$):

- use BFS
- $O(n + m)$ time



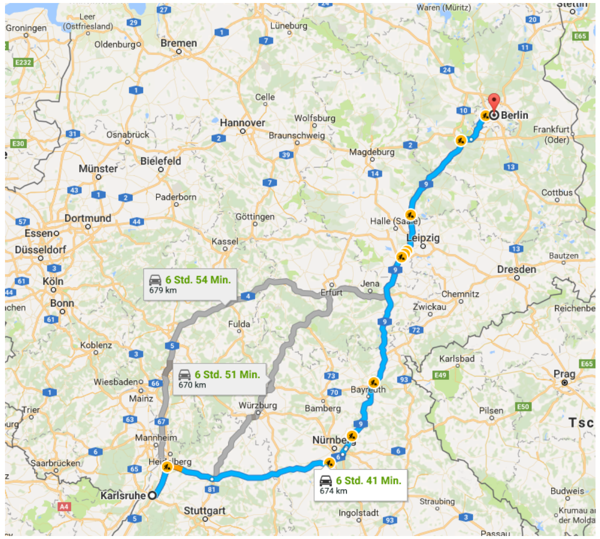
What about weighted graphs?



Shortest Paths

Input:

- Graph $G = (V, E)$
- Edge weights $\omega : E \to \mathbb{R}$
- start node s



Output: $\forall v \in V$: Length $\mu(v)$ of shortest path from s to v

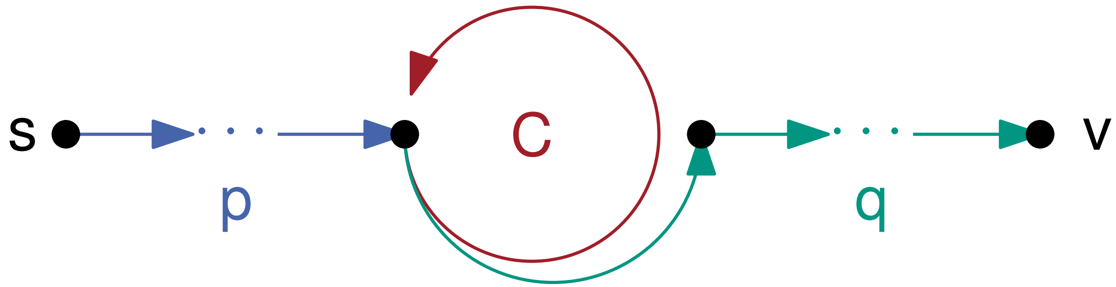
$$\mu(v) := \min\{\omega(p) : p \text{ is path from } s \text{ to } v\}$$

$$\omega(\langle e_1, \ldots, e_k \rangle) := \sum_{i=1}^{k} \omega(e_i)$$

Applications: Route planning, DNA sequencing, production planning,...

Shortest Paths - Basics

Does a shortest path always exist?

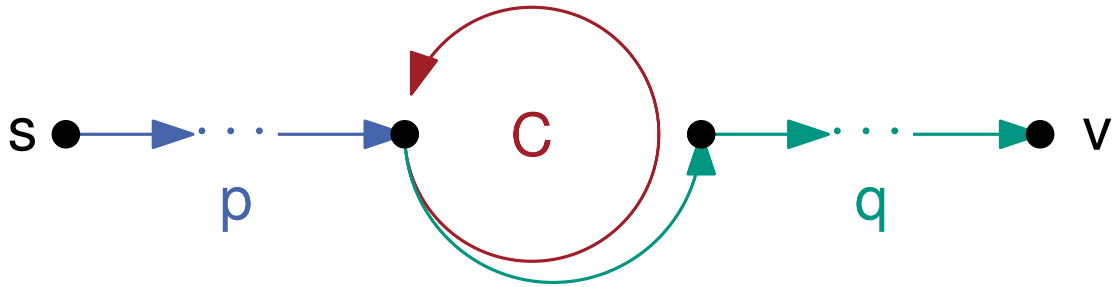


$r = pCq$ is path from s to v

$\Rightarrow$ # paths from s to v is **infinite**: $r^i = pC^iq$

Shortest Paths - Basics

Does a shortest path always exist?



$r = pCq$ is path from s to v

$\Rightarrow$ # paths from s to v is **infinite**: $r^i = pC^iq$

$\Rightarrow$ if C is a **negative** cycle: $\omega(r^{i+1}) < \omega(r^i)$

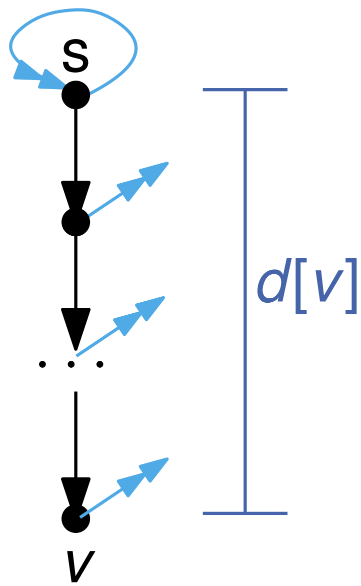
$\omega(C) < 0$

Shortest Paths - Basic Definitions

Assumption: **nonnegative** edge weights $\rightsquigarrow$ no negative cycles

We use 2 Arrays (like in BFS):

- $d[v]$: current (tentative) distance from s to v
  Invariant: $d[v] \geq \mu(v)$
- parent[v]: predecessor of v on (temp.) path from $s \rightsquigarrow v$
- Initialization:
  $d[s] = 0$ parent[s]=s
  $d[v] = \infty$ parent[v]= $\perp$



How to **improve** tentative distance values?

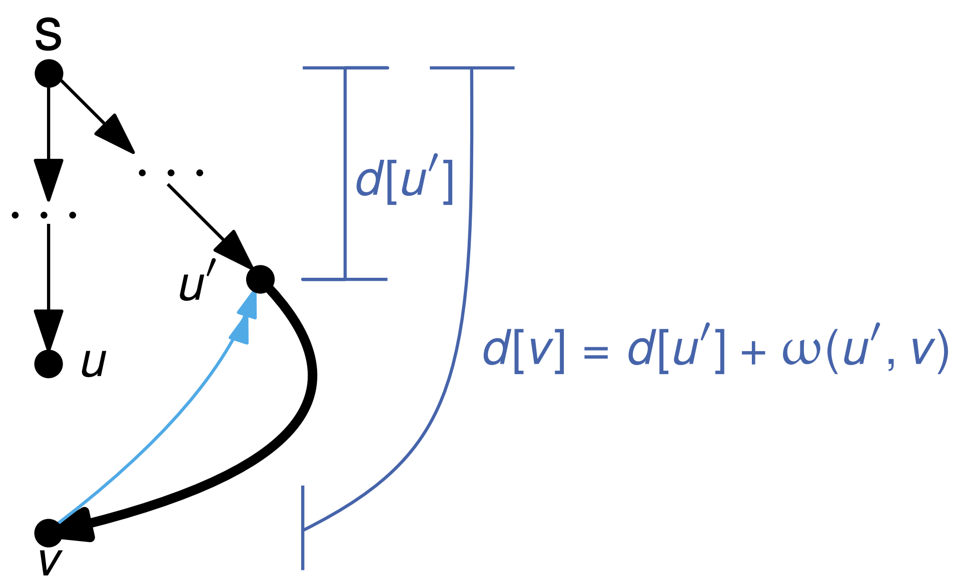
Shortest Paths - Edge Relaxations

Procedure relax($e = (u, v)$: Edge)

if $d[u] + \omega(e) < d[v]$ **then**

$d[v] = d[u] + \omega(e)$

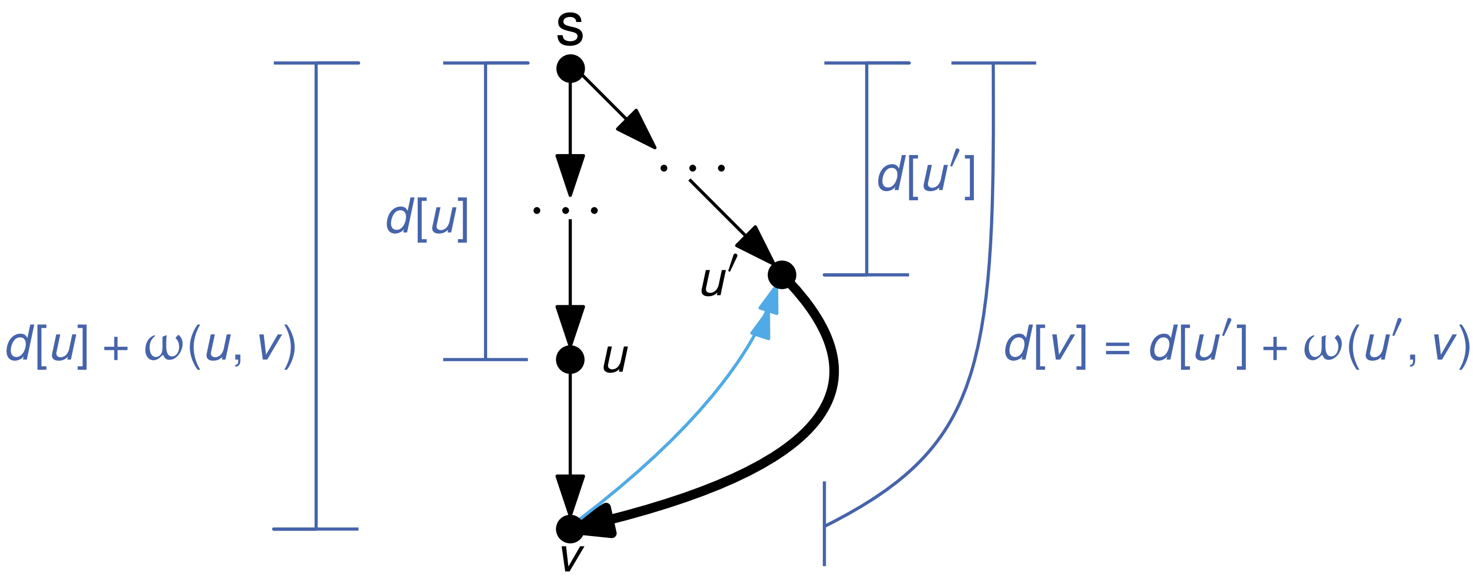
parent[v] = u



Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Edge Relaxations

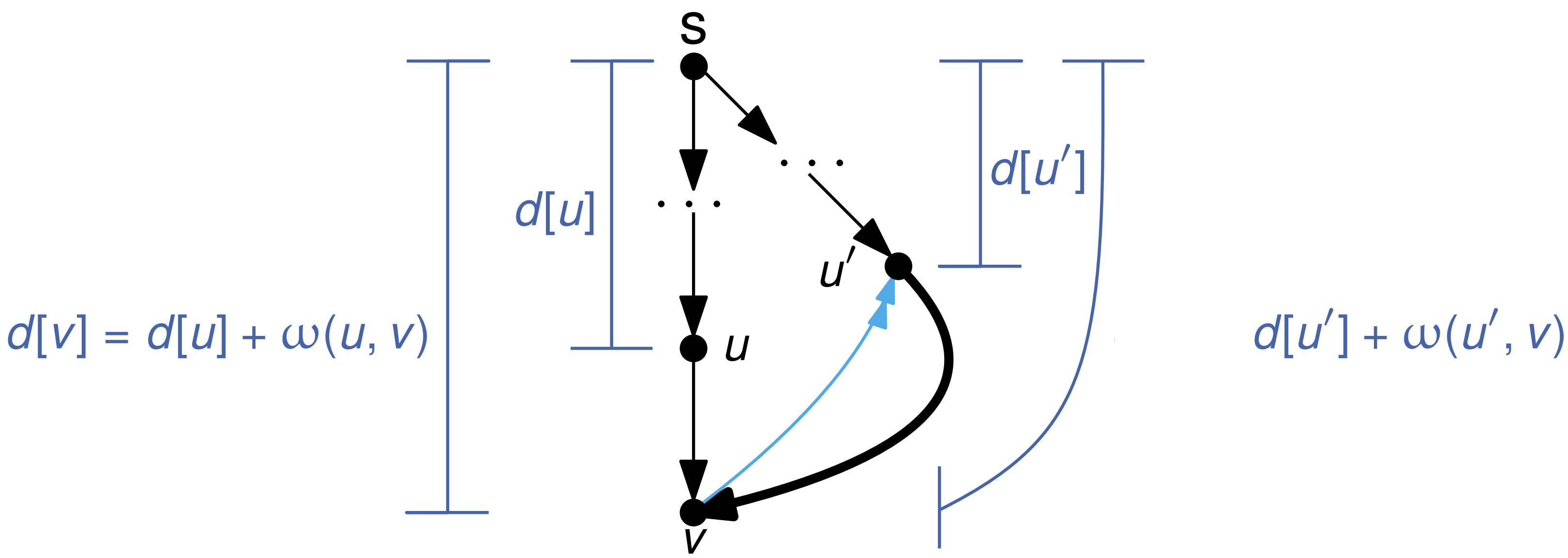
Procedure relax($e = (u, v)$: Edge)
if $d[u] + \omega(e) < d[v]$ **then**
$d[v] = d[u] + \omega(e)$
parent[v] = u



Institute of Theoretical Informatics
Algorithmics Group

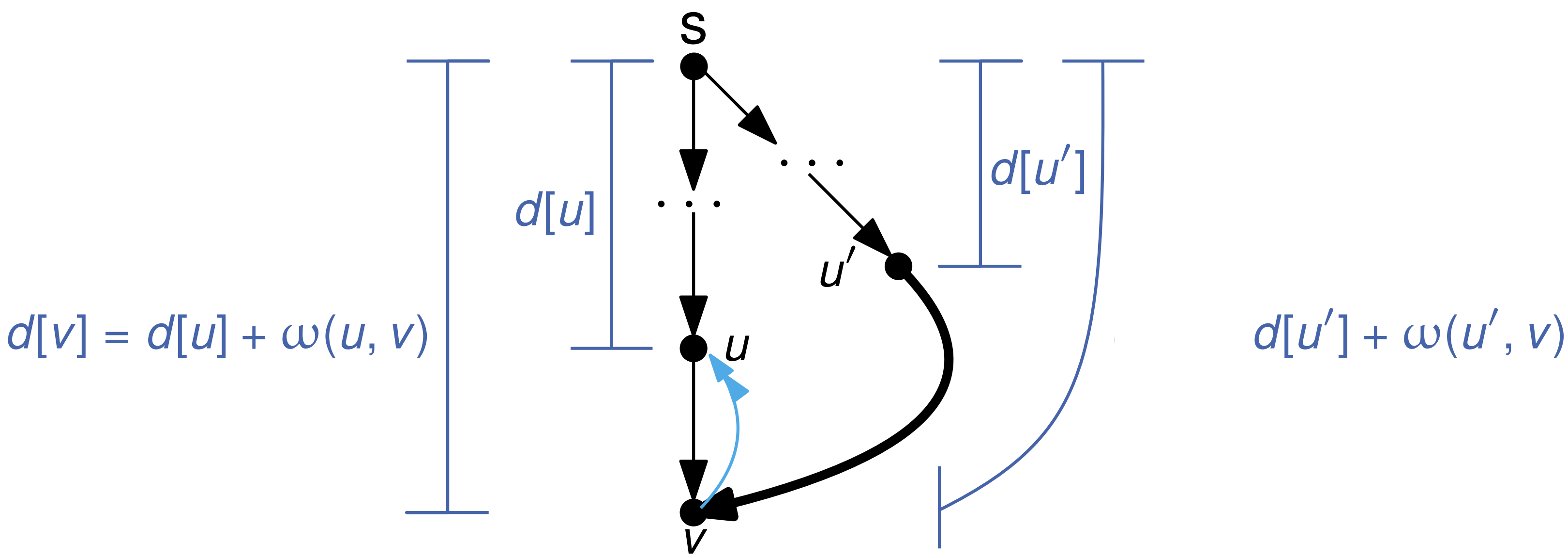
Shortest Paths - Edge Relaxations

Procedure relax($e = (u, v)$: Edge)
if $d[u] + \omega(e) < d[v]$ **then**
$d[v] = d[u] + \omega(e)$
parent[v] = u



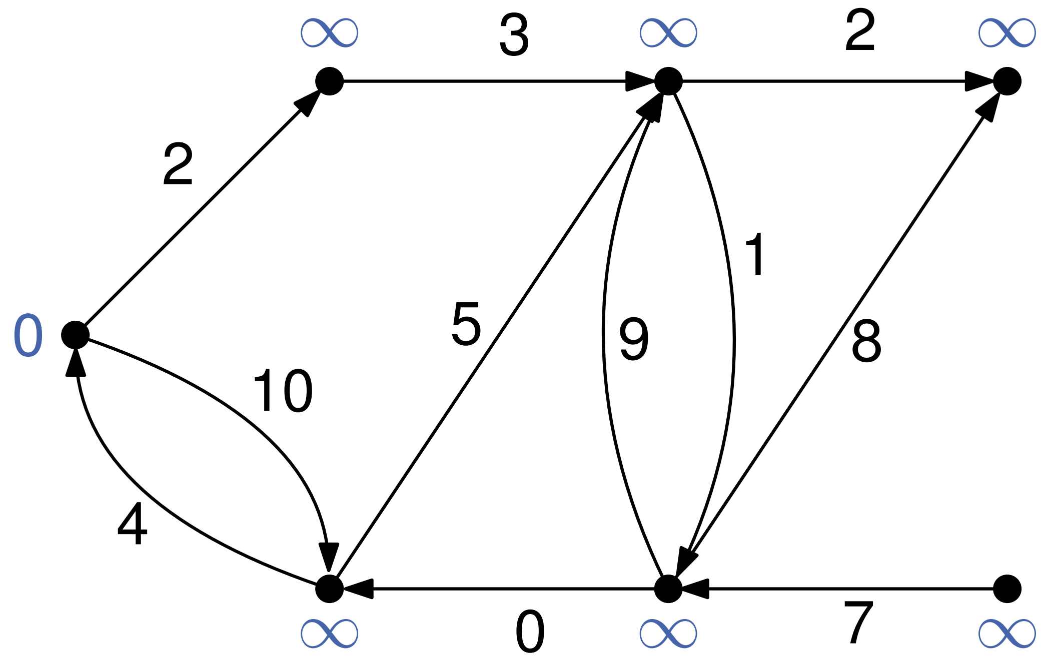
Shortest Paths - Edge Relaxations

Procedure relax($e = (u, v)$: Edge)
if $d[u] + \omega(e) < d[v]$ **then**
$d[v] = d[u] + \omega(e)$
parent[v] = u



Shortest Paths - Dijkstra's Algorithm

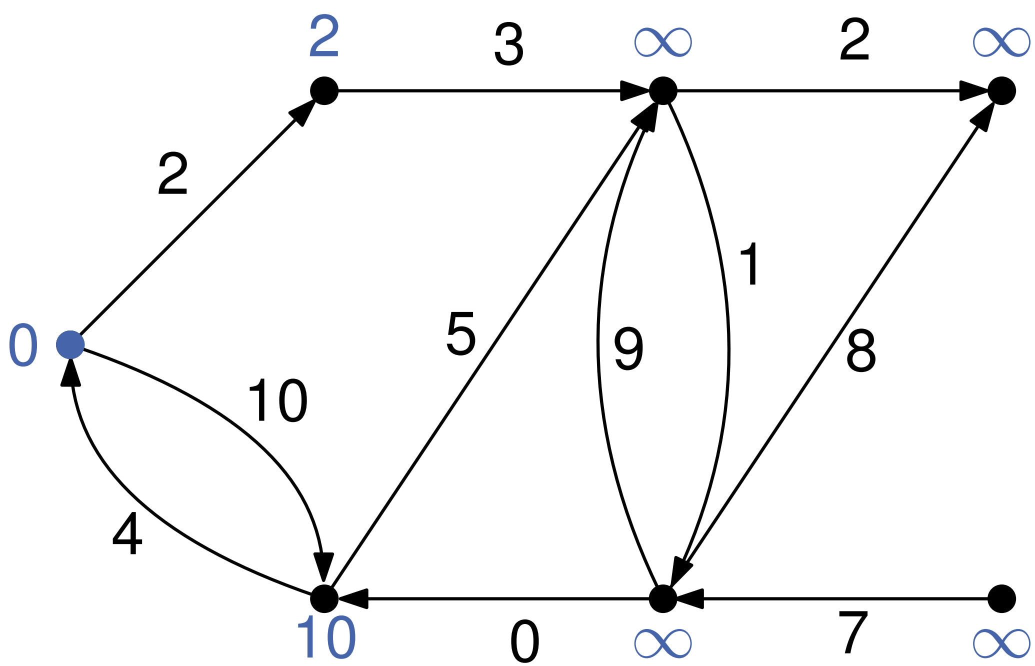
initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    u := non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now



Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Dijkstra's Algorithm

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    u := non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now



Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Dijkstra's Algorithm

Karlsruhe Institute of Technology

initialize d, parent

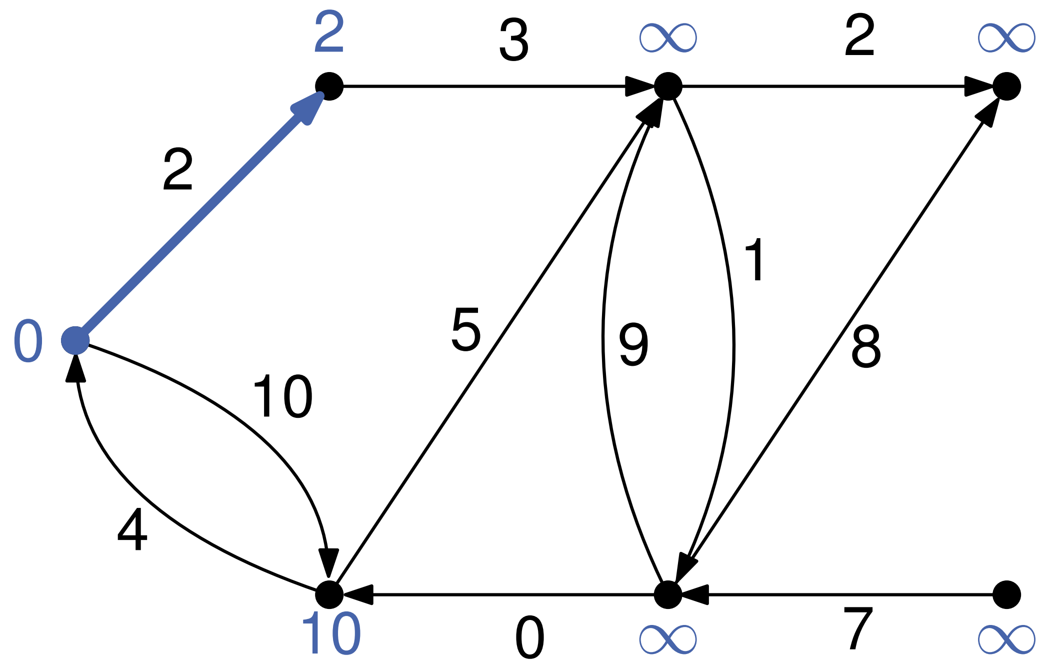
all nodes are non-scanned

while $\exists$ non-scanned node u with $d[u] < \infty$

  u := non-scanned node v with minimal $d[v]$

  relax all edges (u, v) out of u

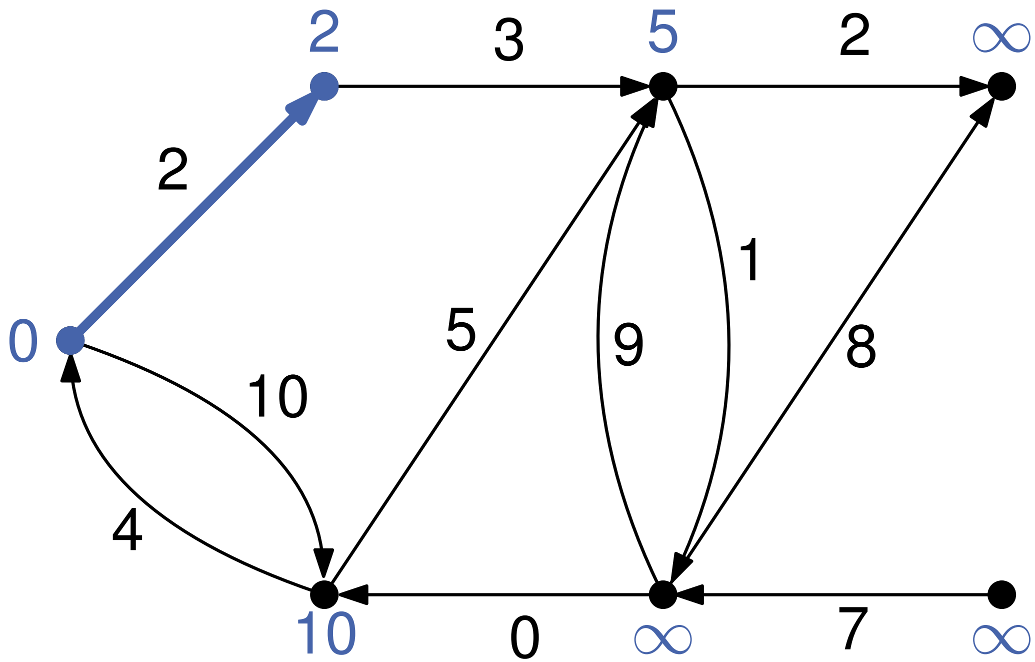
  u is scanned now



Institute of Theoretical Informatics
Algorithmics Group

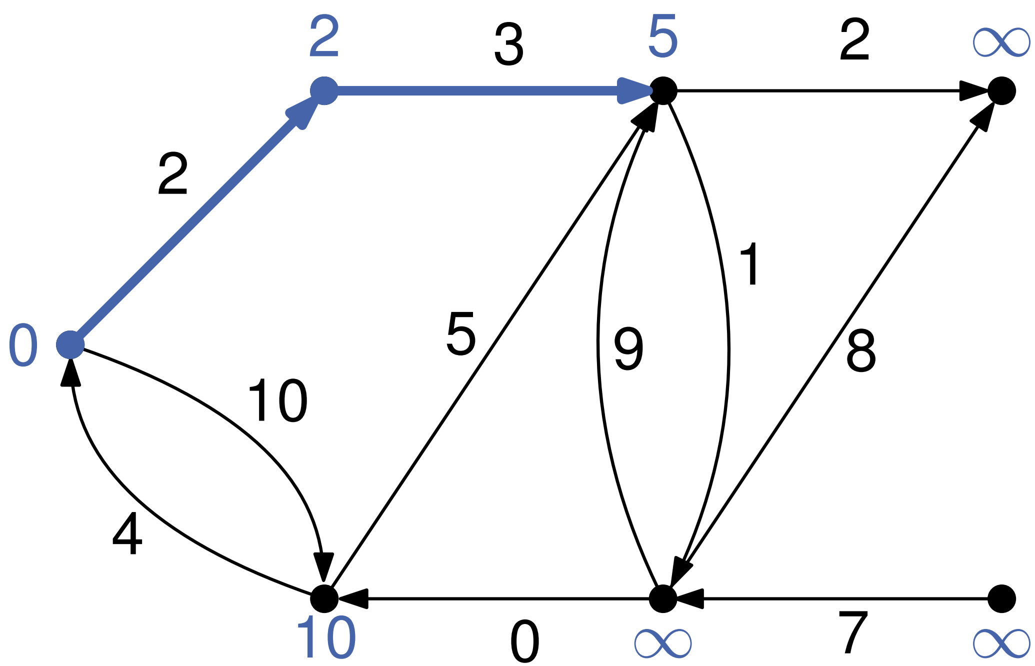
Shortest Paths - Dijkstra's Algorithm

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    $u :=$ non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now



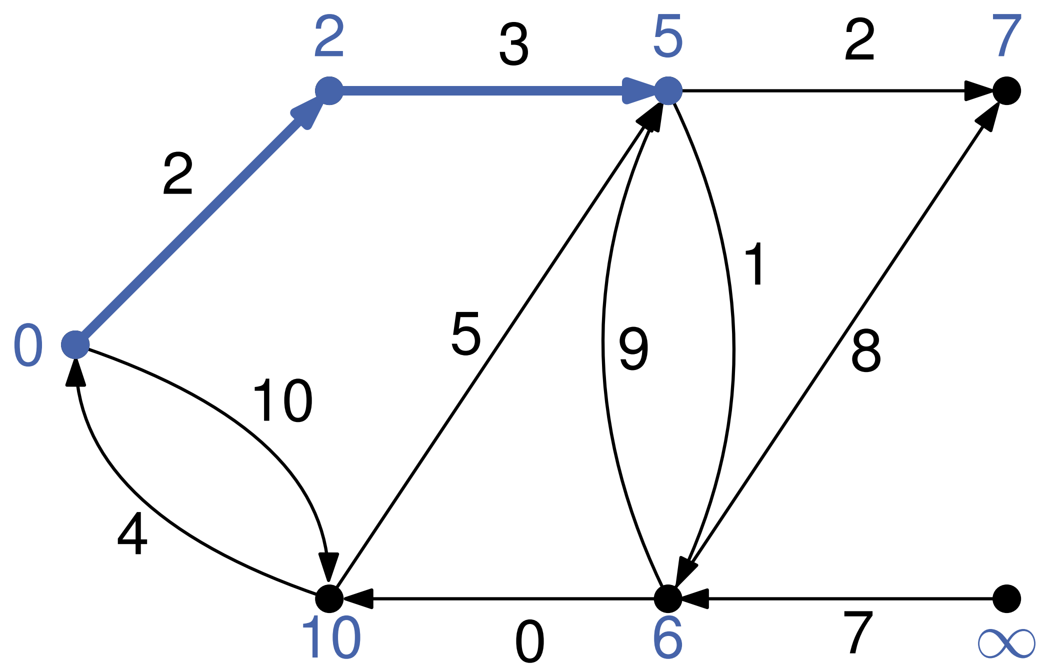
Shortest Paths - Dijkstra's Algorithm

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    u := non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now



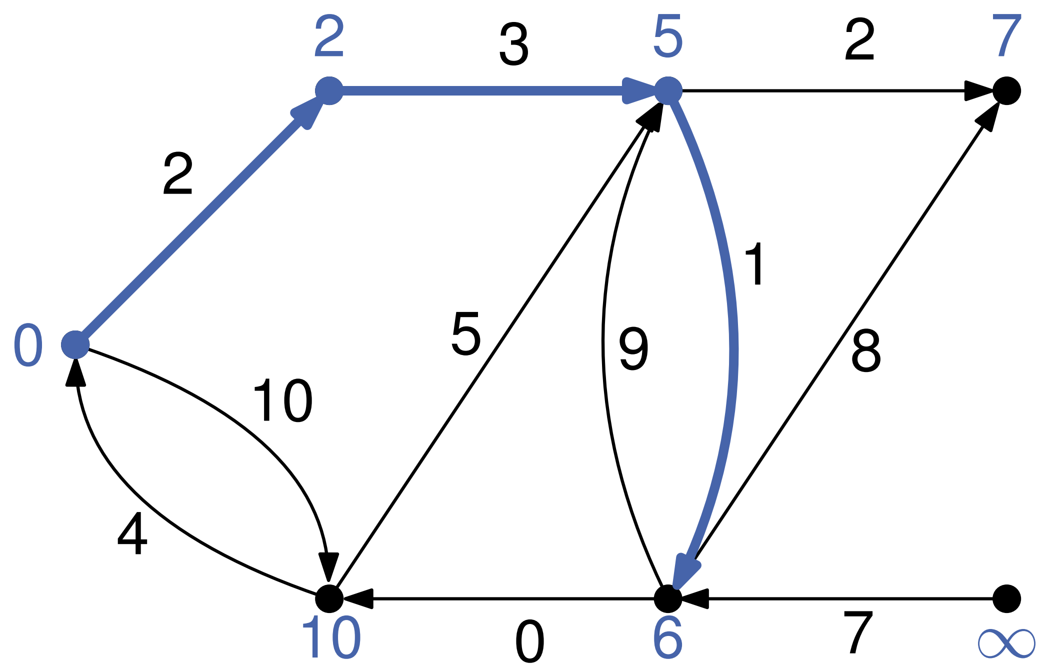
Shortest Paths - Dijkstra's Algorithm

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    u := non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now



Shortest Paths - Dijkstra's Algorithm

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    u := non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now



Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Dijkstra's Algorithm

initialize d, parent

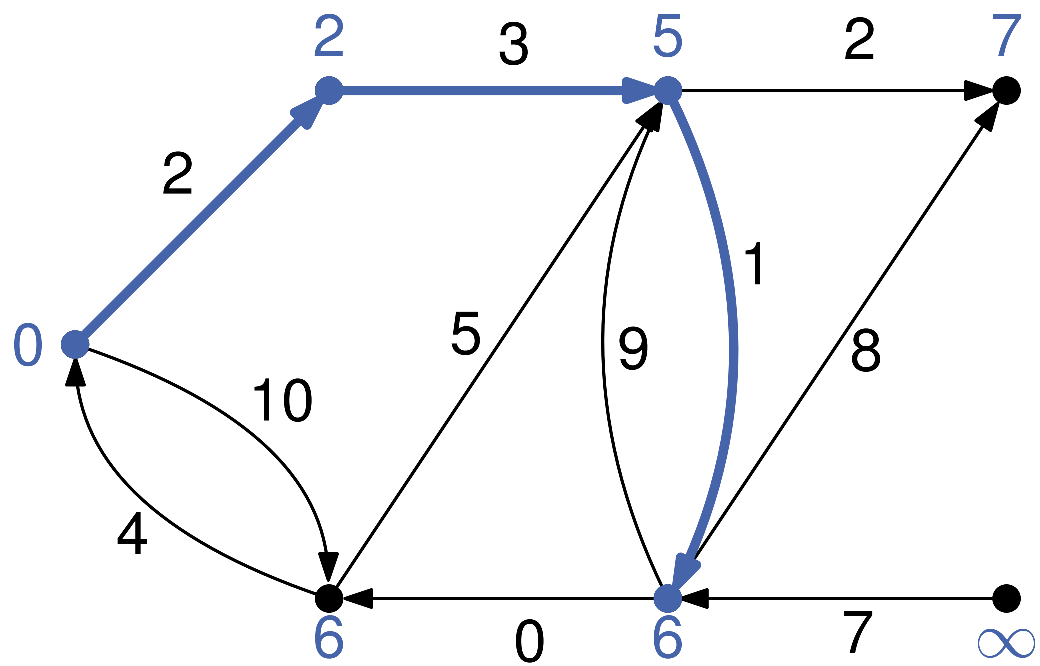
all nodes are non-scanned

while $\exists$ non-scanned node u with $d[u] < \infty$

    u := non-scanned node v with minimal $d[v]$

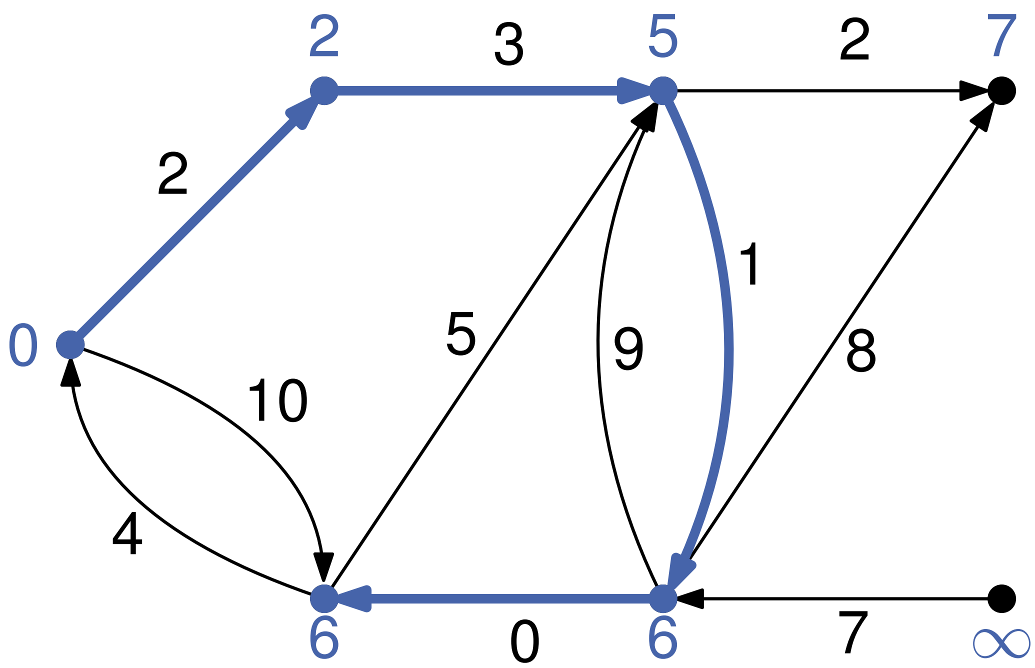
    relax all edges (u, v) out of u

    u is scanned now



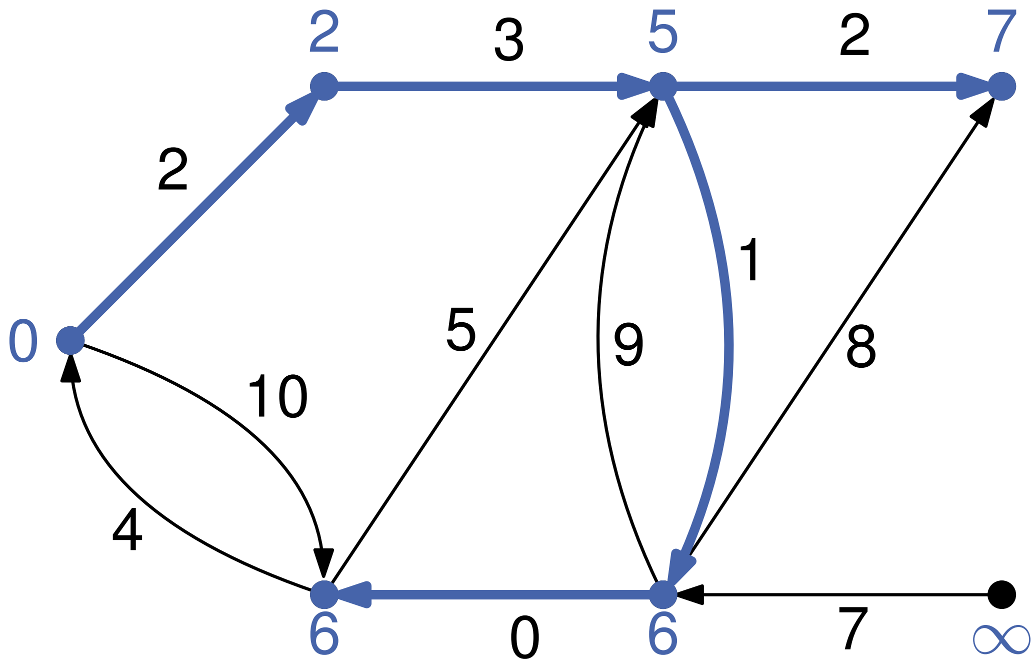
Shortest Paths - Dijkstra's Algorithm

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    u := non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now



Shortest Paths - Dijkstra's Algorithm

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    u := non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now



Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Dijkstra's Algorithm

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is reachable $\rightsquigarrow$ v is scanned
- v is scanned $\rightsquigarrow$ $\mu(v) = d[v]$

Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Dijkstra's Algorithm

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is reachable $\rightsquigarrow$ v is scanned

Assumption:
reachable from s, but

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scanned}} \longrightarrow \overbrace{v_i}^{\text{unscanned}} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{never scanned}}}_{\text{a shortest } s\text{–}v \text{ path}}$$

Shortest Paths - Dijkstra's Algorithm

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is reachable $\rightsquigarrow$ v is scanned

Assumption:
reachable from s, but

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scanned}} \longrightarrow \overbrace{v_i}^{\text{unscanned}} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{never scanned}}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow i > 1$, because s is scanned

Shortest Paths - Dijkstra's Algorithm

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is reachable $\rightsquigarrow$ v is scanned

Assumption:
reachable from s, but

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scanned}} \longrightarrow \overbrace{v_i}^{\text{unscanned}} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{never scanned}}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow$ $i > 1$, because s is scanned
$\Rightarrow$ v_{i-1} has been scanned (by definition)

Shortest Paths - Dijkstra's Algorithm

Karlsruhe Institute of Technology

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is reachable $\rightsquigarrow$ v is scanned

Assumption:
reachable from s, but

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scanned}} \longrightarrow \overbrace{v_i}^{\text{unscanned}} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{never scanned}}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow$ $i > 1$, because s is scanned
$\Rightarrow$ v_{i-1} has been scanned (by definition)
$\Rightarrow$ edge $v_{i-1} \to v_i$ was relaxed

Shortest Paths - Dijkstra's Algorithm

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is reachable $\rightsquigarrow$ v is scanned

Assumption:
reachable from s, but

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scanned}} \longrightarrow \overbrace{v_i}^{\text{unscanned}} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{never scanned}}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow$ $i > 1$, because s is scanned
$\Rightarrow$ v_{i-1} has been scanned (by definition)
$\Rightarrow$ edge $v_{i-1} \to v_i$ was relaxed
$\Rightarrow$ $d[v_i] < \infty$
$\Rightarrow$ **contradiction**: only nodes x with $d[x] = \infty$ remain unscanned

Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Dijkstra's Algorithm

Karlsruhe Institute of Technology

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is scanned $\rightsquigarrow \mu(v) = d[v]$

Shortest Paths - Dijkstra's Algorithm

Karlsruhe Institute of Technology

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is scanned $\rightsquigarrow \mu(v) = d[v]$

Assumption:
v is first scanned node with $\mu(v) < d[v]$

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scan time} < t} \longrightarrow \overbrace{v_i}^{\text{scan time} \geq t} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{scan time} = t}}_{\text{a shortest } s\text{–}v \text{ path}}$$

Shortest Paths - Dijkstra's Algorithm

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is scanned $\rightsquigarrow \mu(v) = d[v]$

Assumption:
v is first scanned node with $\mu(v) < d[v]$

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scan time} < t} \longrightarrow \overbrace{v_i}^{\text{scan time} \geq t} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{scan time} = t}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow v_{i-1}$ was scanned before t (by definition)

Shortest Paths - Dijkstra's Algorithm

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is scanned $\rightsquigarrow \mu(v) = d[v]$

Assumption:
v is first scanned node with $\mu(v) < d[v]$

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scan time} < t} \longrightarrow \overbrace{v_i}^{\text{scan time} \geq t} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{scan time} = t}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow$ v_{i-1} was scanned before t (by definition)
$\Rightarrow$ $d[v_{i-1}] = \mu(v_{i-1})$ when v_{i-1} is scanned

Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Dijkstra's Algorithm

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is scanned $\rightsquigarrow \mu(v) = d[v]$

Assumption:
v is first scanned node with $\mu(v) < d[v]$

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scan time} < t} \longrightarrow \overbrace{v_i}^{\text{scan time} \geq t} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{scan time} = t}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow v_{i-1}$ was scanned before t (by definition)
$\Rightarrow d[v_{i-1}] = \mu(v_{i-1})$ when v_{i-1} is scanned
$\Rightarrow$ edge $v_{i-1} \to v_i$ was relaxed

Institute of Theoretical Informatics
Algorithmics Group

Shortest Paths - Dijkstra's Algorithm

Karlsruhe Institute of Technology

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is scanned $\leadsto \mu(v) = d[v]$

Assumption:
v is first scanned node with $\mu(v) < d[v]$

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scan time} < t} \longrightarrow \overbrace{v_i}^{\text{scan time} \geq t} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{scan time} = t}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow$ v_{i-1} was scanned before t (by definition)
$\Rightarrow$ $d[v_{i-1}] = \mu(v_{i-1})$ when v_{i-1} is scanned
$\Rightarrow$ edge $v_{i-1} \rightarrow v_i$ was relaxed
$\Rightarrow$ $d[v_i] = d[v_{i-1}] + \omega(v_{i-1}, v_i) = \mu(v_{i-1}) + \omega(v_{i-1}, v_i) = \mu(v_i)$

Shortest Paths - Dijkstra's Algorithm

Karlsruhe Institute of Technology

Theorem:
Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.

Proof: We show: $\forall v \in V$:

- v is scanned $\rightsquigarrow \mu(v) = d[v]$

Assumption:
v is first scanned node with $\mu(v) < d[v]$

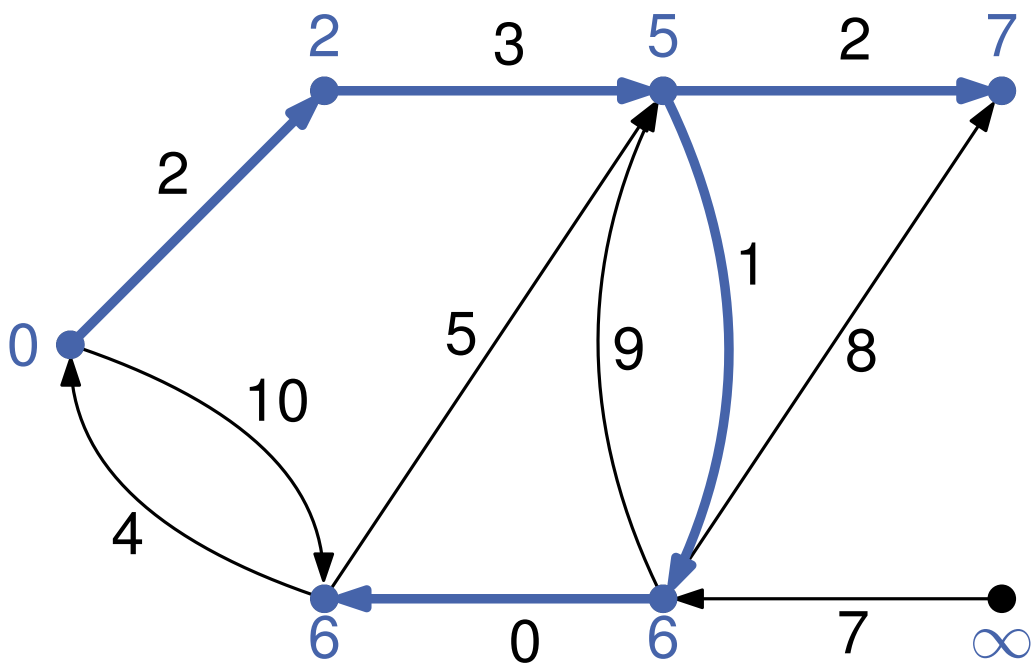
$$\underbrace{\overbrace{s = v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{scan time} < t} \longrightarrow \overbrace{v_i}^{\text{scan time} \geq t} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{scan time} = t}}_{\text{a shortest } s\text{–}v \text{ path}}$$

$\Rightarrow$ v_{i-1} was scanned before t (by definition)
$\Rightarrow$ $d[v_{i-1}] = \mu(v_{i-1})$ when v_{i-1} is scanned
$\Rightarrow$ edge $v_{i-1} \to v_i$ was relaxed
$\Rightarrow$ $d[v_i] = d[v_{i-1}] + \omega(v_{i-1}, v_i) = \mu(v_{i-1}) + \omega(v_{i-1}, v_i) = \mu(v_i)$
$\Rightarrow$ at time t : $d[v_i] = \mu(v_i) \leq \mu(v) < d[v]$
$\Rightarrow$ v_i is scanned before v! **contradiction!**

Dijkstra's Algorithm - Implementation

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    u := non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now

How?



Dijkstra's Algorithm - Implementation

Function Dijkstra$(s: \text{NodeId}): \text{NodeArray} \times \text{NodeArray}$

```
    d = {∞, ..., ∞}; parent[s]:= s; d[s] := 0;   Q.insert(s)      // O(n)
    while Q ≠ ∅ do
        u := Q.deleteMin                                          // ≤ n×
        foreach edge e = (u, v) ∈ E do                            // ≤ m×
            if d[u] + c(e) < d[v] then                            // ≤ m×
                d[v]:= d[u] + c(e)                                // ≤ m×
                parent[v] := u                                    // ≤ m×
                if v ∈ Q then Q.decreaseKey(v)                    // ≤ m×
                else Q.insert(v)                                  // ≤ n×
    return (d, parent)
```

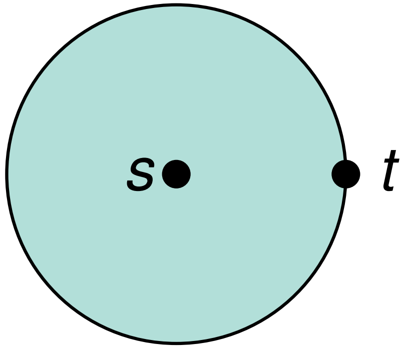
Total Running Time:

$$T_{\text{Dijkstra}} = \mathrm{O}\left(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\right)$$

Goal-Directed Search / Pathfinding

Goal: Find distance from s to a specific node t

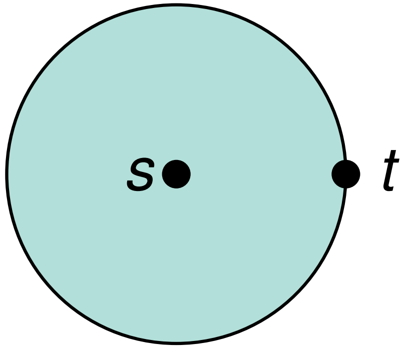
One Solution:
stop Dijkstra as soon as t is removed from PQ



Institute of Theoretical Informatics
Algorithmics Group

Goal-Directed Search / Pathfinding

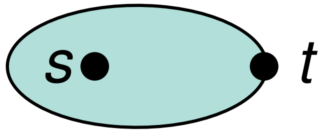
Goal: Find distance from s to a specific node t



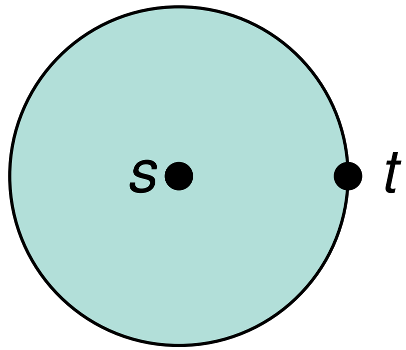
One Solution:
stop Dijkstra as soon as t is removed from PQ

A* Search:

- Idea: bias search towards the target
- $\forall v \in V$: heuristic $f(v)$ estimates distance $\mu(v, t)$
- modified distance fct. $\forall e = (u, v) \in E : \overline{c} = c(e) + f(v) - f(u)$



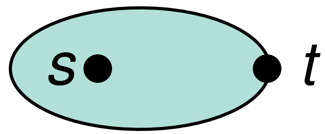
Goal-Directed Search / Pathfinding



Goal: Find distance from s to a specific node t

One Solution:
stop Dijkstra as soon as t is removed from PQ

A* Search:



- Idea: bias search towards the target
- $\forall v \in V$: heuristic $f(v)$ estimates distance $\mu(v, t)$
- modified distance fct. $\forall e = (u, v) \in E : \bar{c} = c(e) + f(v) - f(u)$

Optimistic Example: $f(v) = \mu(v, t)$

$\Rightarrow \bar{c}(u, v) = c(u, v) + \mu(v, t) - \mu(u, t) = 0$ if (u, v) is on shortest s, t path

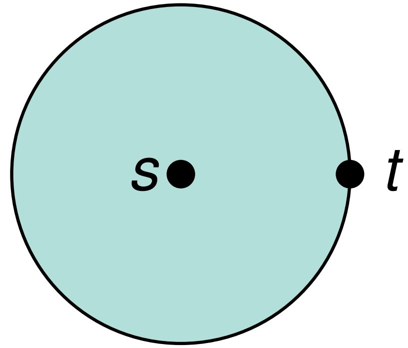
$$s \to \cdots \to \underbrace{u \xrightarrow{c(u,v)} \overbrace{v \to \cdots \to t}^{\mu(v,t)}}_{\mu(u,t)}$$

$\Rightarrow$ Dijkstra only scans nodes along shortest path!

Institute of Theoretical Informatics
Algorithmics Group

Goal-Directed Search / Pathfinding

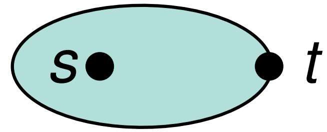
Goal: Find distance from s to a specific node t



One Solution:
stop Dijkstra as soon as t is removed from PQ

A* Search:

- Idea: bias search towards the target
- $\forall v \in V$: heuristic $f(v)$ estimates distance $\mu(v, t)$
- modified distance fct. $\forall e = (u, v) \in E : \bar{c} = c(e) + f(v) - f(u)$



Optimistic Example: $f(v) = \mu(v, t)$

$\Rightarrow \bar{c}(u, v) = c(u, v) + \mu(v, t) - \mu(u, t) = 0$ if (u, v) is on shortest s, t path

$$s \to \cdots \to \underbrace{u \xrightarrow{c(u,v)} \overbrace{v \to \cdots \to t}^{\mu(v,t)}}_{\mu(u,t)}$$

$\Rightarrow$ Dijkstra only scans nodes along shortest path!

Interactive Demo: `http://www.ryanpon.com/animate`

More on Shortest Paths

- DAGs:
  $\Rightarrow$ relax edges in topological order of vertices: $O(m+n)$
- arbitrary edge weights:
  $\Rightarrow$ Bellman-Ford Algorithm (Idea: relax all edges $n-1$ times): $O(m\, n)$
- All-Pairs Shortest Paths
  - dense graphs (without negative cycles)
    $\Rightarrow$ Floyd–Warshall Algorithm: $O\left(n^3\right)$
  - non-negative edge weights:
    $\Rightarrow n \times$ Dijkstra: $O(n(m+n\log n))$
  - arbitrary edge weights:
    $\Rightarrow n \times$ Bellman-Ford: $O\left(n^2\, m\right)$
    $\Rightarrow 1\times$ Bellman-Ford + $n \times$ Dijkstra: $O(n(m+n\log n))$ [1]

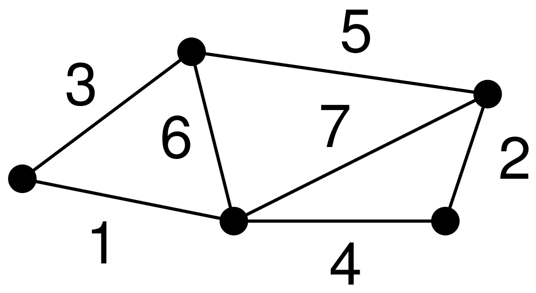
[1] K. Mehlhorn, V. Priebe, G. Schäfer, N. Sivadasan: All-pairs shortest-paths computation in the presence of negative cycles. Inf. Process. Lett. 81(6): 341-343 (2002)

Minimal Spanning Tree (MST)

Given undirected Graph $G = (V, E)$ with edge weights $c(e) \in \mathcal{R}_+$

- G connected

$\Rightarrow$ Find a tree (V,T) with minimal weight $\sum_{e \in T} c(e)$ that connects all vertices



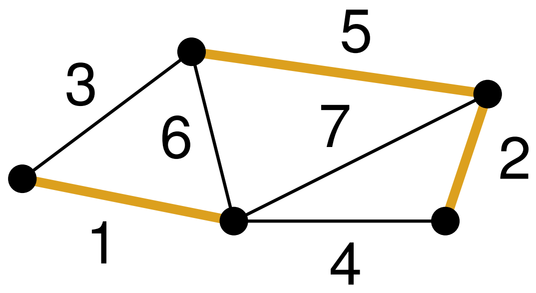
Institute of Theoretical Informatics
Algorithmics Group

Minimal Spanning Tree (MST)

Given undirected Graph $G = (V, E)$ with edge weights $c(e) \in \mathcal{R}_+$

- G connected

$\Rightarrow$ Find a tree (V,T) with minimal weight $\sum_{e \in T} c(e)$ that connects all vertices



Vertices unconnected

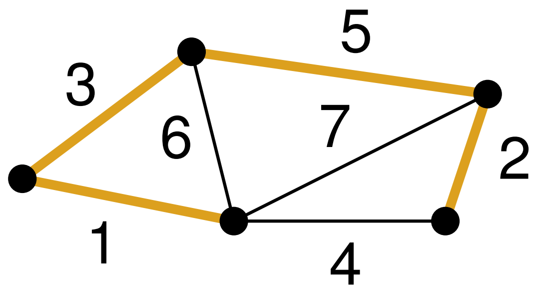
Non-minimal weight

Minimal Spanning Tree (MST)

Given undirected Graph $G = (V, E)$ with edge weights $c(e) \in \mathcal{R}_+$

- G connected

$\Rightarrow$ Find a tree (V,T) with minimal weight $\sum_{e \in T} c(e)$ that connects all vertices



Vertices connected

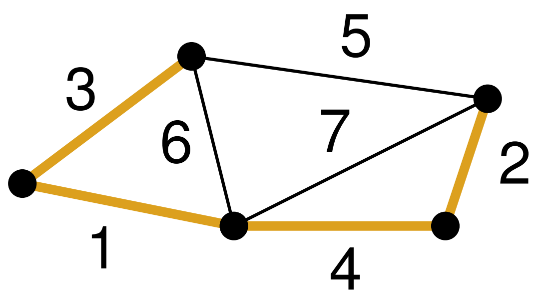
Non-minimal weight

Minimal Spanning Tree (MST)

Given undirected Graph $G = (V, E)$ with edge weights $c(e) \in \mathcal{R}_+$

- G connected

$\Rightarrow$ Find a tree (V,T) with minimal weight $\sum_{e \in T} c(e)$ that connects all vertices



Vertices connected

Minimal weight

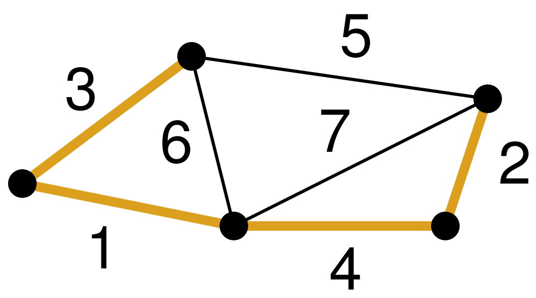
Institute of Theoretical Informatics
Algorithmics Group

Minimal Spanning Tree (MST)

Given undirected Graph $G = (V, E)$ with edge weights $c(e) \in \mathcal{R}_+$

- G connected

$\Rightarrow$ Find a tree (V,T) with minimal weight $\sum_{e \in T} c(e)$ that connects all vertices

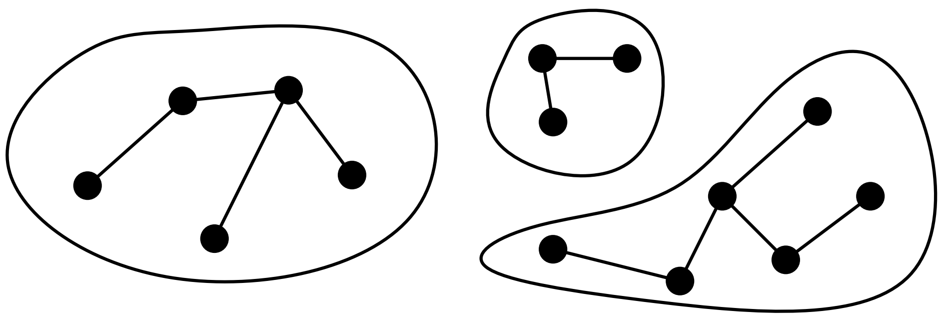


Vertices connected

Minimal weight

- G unconnected

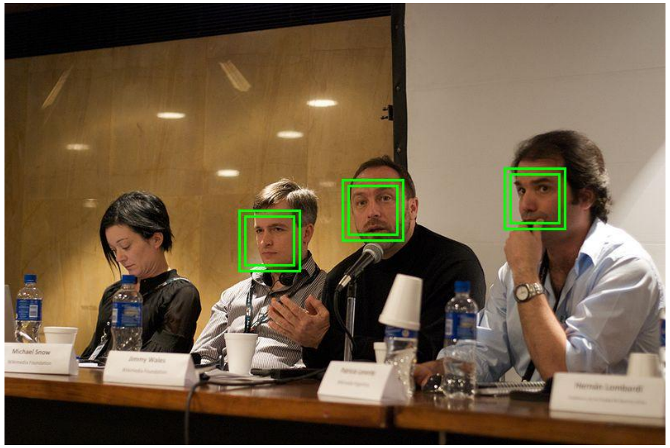
Find minimal spanning forest (MSF) that spans all connected components



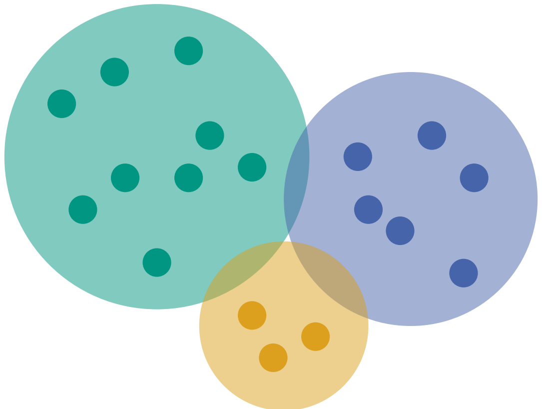
Applications

Network design

Reduce storage for protein sequencing



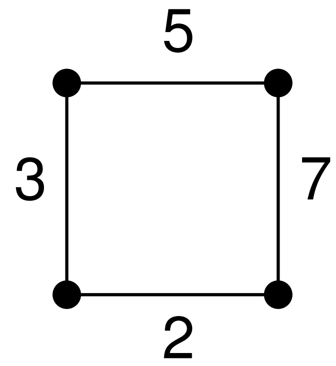
Learning features for face verification



Cluster analysis

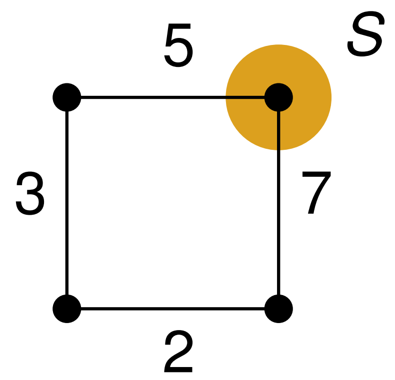
Finding MST Edges

- Cut property
  - Arbitrary subset $S \subset V$
  - Cut edges $C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$



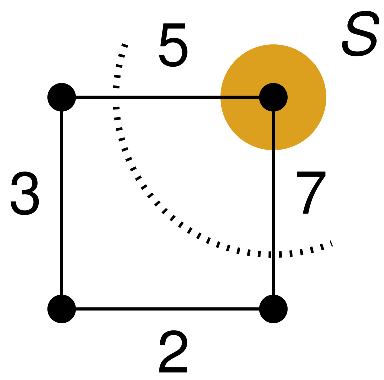
Finding MST Edges

- Cut property
  - Arbitrary subset $S \subset V$
  - Cut edges $C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$



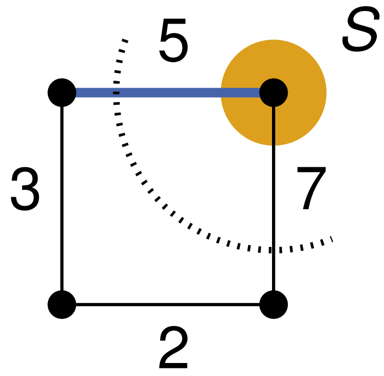
Finding MST Edges

- Cut property
  - Arbitrary subset $S \subset V$
  - Cut edges $C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$



Finding MST Edges

- Cut property
  - Arbitrary subset $S \subset V$
  - Cut edges $C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$

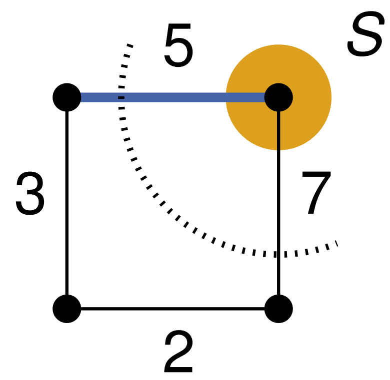


$\Rightarrow$ Lightest edge in C can be used in an MST
(Proof via exchange with heavier cycle edge)

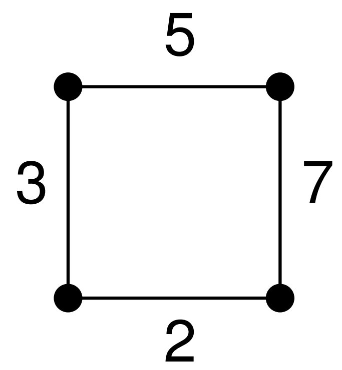
Finding MST Edges

- Cut property
  - Arbitrary subset $S \subset V$
  - Cut edges $C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$

$\Rightarrow$ Lightest edge in C can be used in an MST
(Proof via exchange with heavier cycle edge)



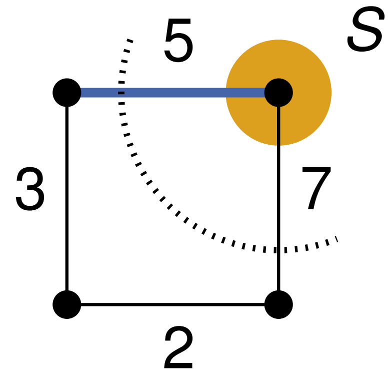
- Cycle property
  - Arbitrary cycle C in G



Institute of Theoretical Informatics
Algorithmics Group

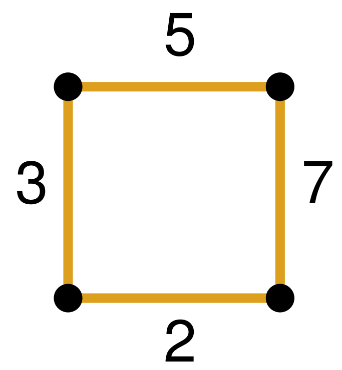
Finding MST Edges

- Cut property
  - Arbitrary subset $S \subset V$
  - Cut edges $C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$



$\Rightarrow$ Lightest edge in C can be used in an MST
(Proof via exchange with heavier cycle edge)

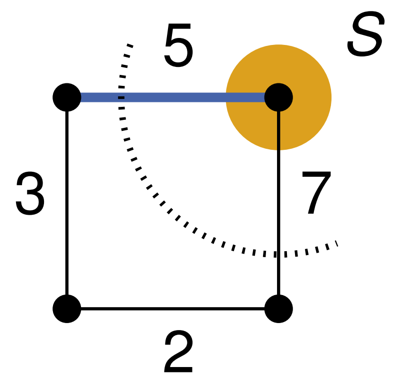
- Cycle property
  - Arbitrary cycle C in G



Institute of Theoretical Informatics
Algorithmics Group

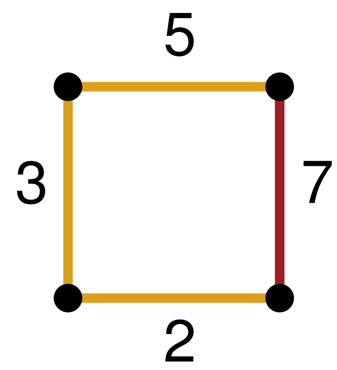
Finding MST Edges

- Cut property
  - Arbitrary subset $S \subset V$
  - Cut edges $C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$



$\Rightarrow$ Lightest edge in C can be used in an MST
(Proof via exchange with heavier cycle edge)

- Cycle property
  - Arbitrary cycle C in G

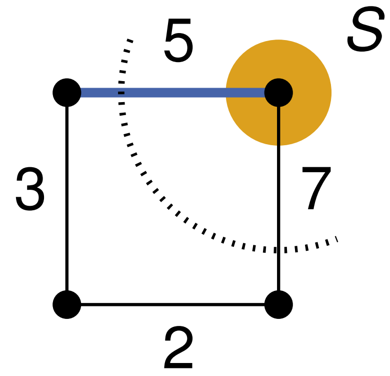


$\Rightarrow$ Heaviest edge in C is not needed in an MST
(Proof via exchange with lighter cycle edge)

Institute of Theoretical Informatics
Algorithmics Group

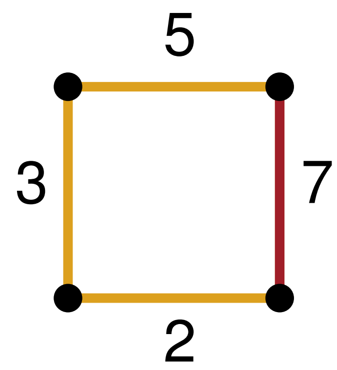
Finding MST Edges

- Cut property
  - Arbitrary subset $S \subset V$
  - Cut edges $C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$



$\Rightarrow$ Lightest edge in C can be used in an MST
(Proof via exchange with heavier cycle edge)

- Cycle property
  - Arbitrary cycle C in G



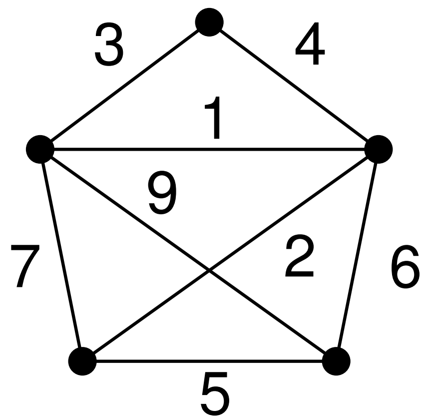
$\Rightarrow$ Heaviest edge in C is not needed in an MST
(Proof via exchange with lighter cycle edge)

Essential properties for developing MST algorithms

Jarnik-Prim Algorithm

Use cut property to gradually grow the MST

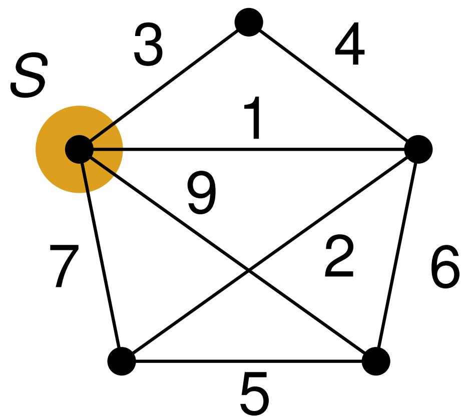
1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$



Jarnik-Prim Algorithm

Use cut property to gradually grow the MST

1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$

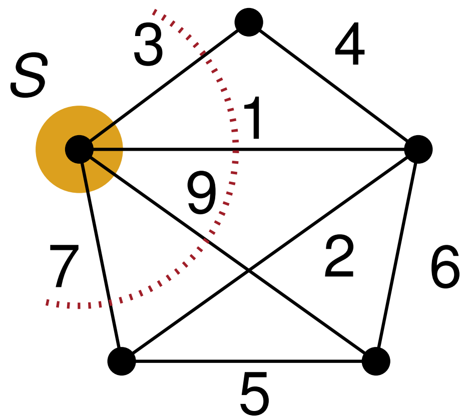


Jarnik-Prim Algorithm

Karlsruhe Institute of Technology

Use cut property to gradually grow the MST

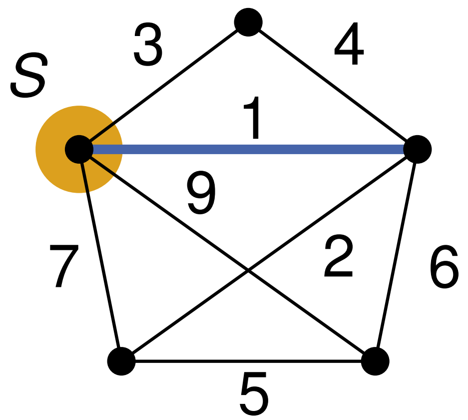
1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$



Jarnik-Prim Algorithm

Use cut property to gradually grow the MST

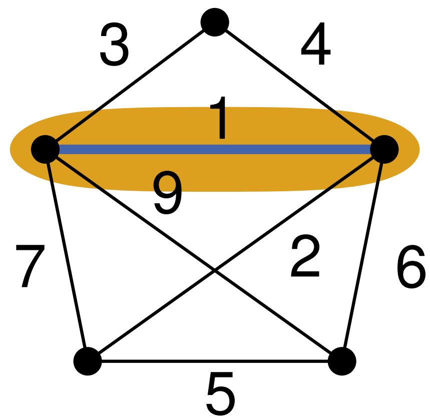
1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$



Jarnik-Prim Algorithm

Use cut property to gradually grow the MST

1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$

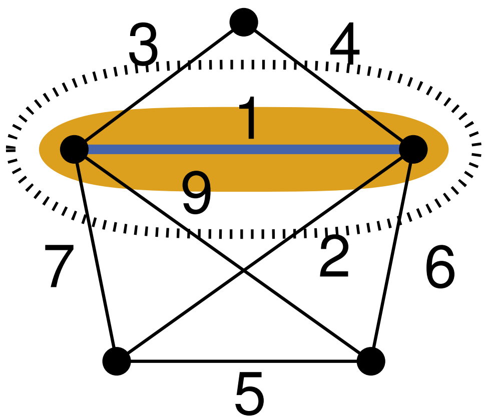


Institute of Theoretical Informatics
Algorithmics Group

Jarnik-Prim Algorithm

Use cut property to gradually grow the MST

1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$



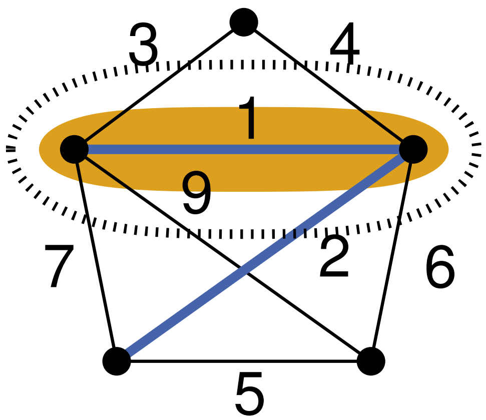
Institute of Theoretical Informatics
Algorithmics Group

Jarnik-Prim Algorithm

Karlsruhe Institute of Technology

Use cut property to gradually grow the MST

1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$

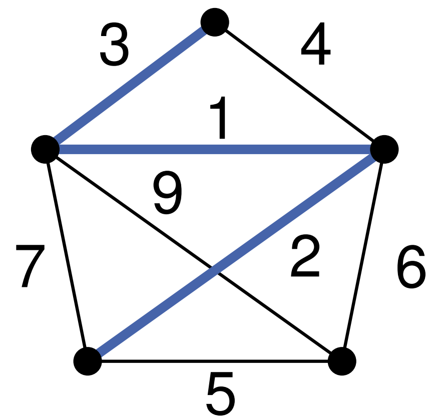


Institute of Theoretical Informatics
Algorithmics Group

Jarnik-Prim Algorithm

Use cut property to gradually grow the MST

1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$

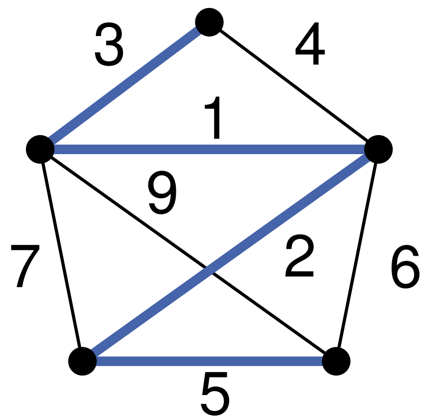


Institute of Theoretical Informatics
Algorithmics Group

Jarnik-Prim Algorithm

Use cut property to gradually grow the MST

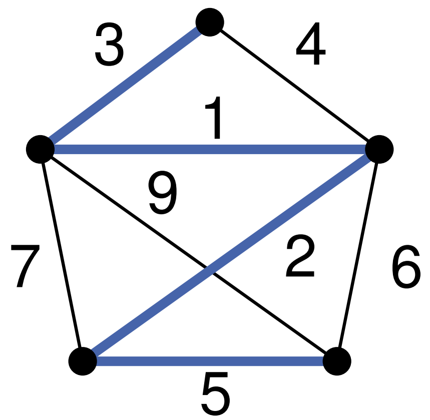
1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$



Jarnik-Prim Algorithm

Use cut property to gradually grow the MST

1. Start with empty MST T
2. Select random start vertex $S = \{s\}$
3. Repeat $n - 1$ times
   (a) Find edge $\{u, v\}$ fulfilling cut property for S
   (b) $S = S \cup \{v\}$
   (c) $T = T \cup \{\{u, v\}\}$



$\Rightarrow$ Lightest edge using PQ

Good $\mathcal{O}(m + n \log n)$

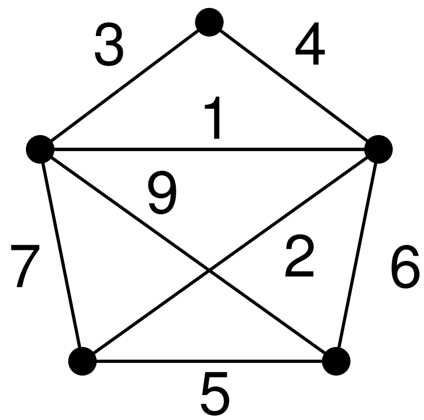
using Fibonacci Heaps

Institute of Theoretical Informatics
Algorithmics Group

Kruskal's Algorithm

Use cut and cycle property to merge subtrees of MST

1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)

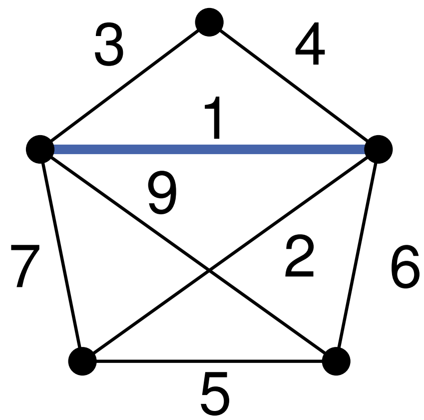


Institute of Theoretical Informatics
Algorithmics Group

Kruskal's Algorithm

Use cut and cycle property to merge subtrees of MST

1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)

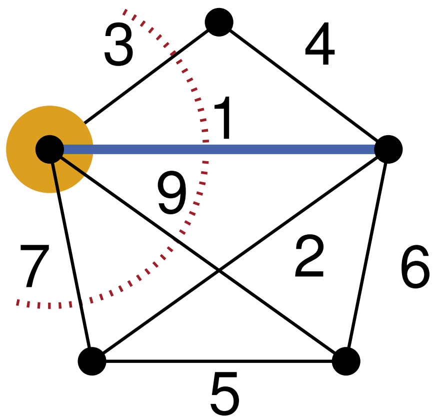


Kruskal's Algorithm

Karlsruhe Institute of Technology

Use cut and cycle property to merge subtrees of MST

1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)

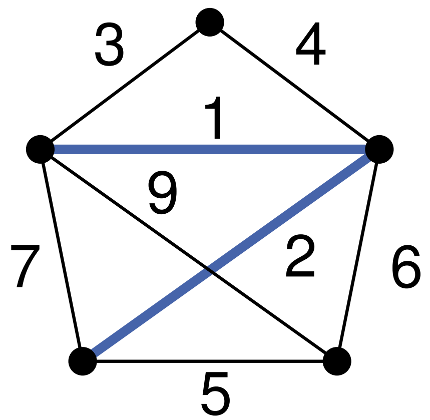


Institute of Theoretical Informatics
Algorithmics Group

Kruskal's Algorithm

Use cut and cycle property to merge subtrees of MST

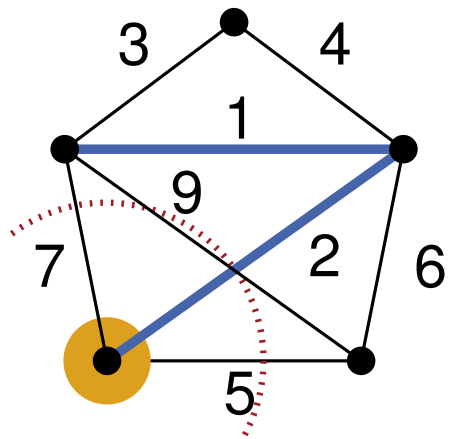
1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)



Kruskal's Algorithm

Use cut and cycle property to merge subtrees of MST

1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)

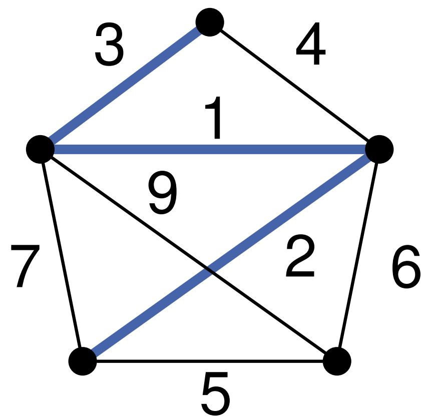


Institute of Theoretical Informatics
Algorithmics Group

Kruskal's Algorithm

Use cut and cycle property to merge subtrees of MST

1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)



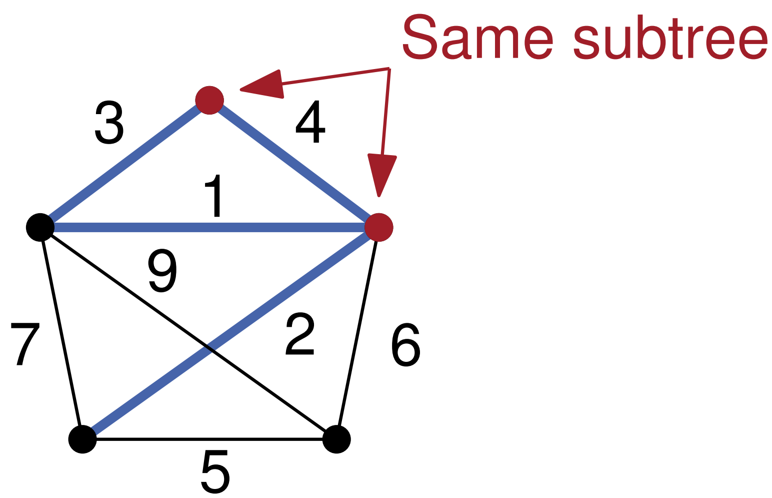
Institute of Theoretical Informatics
Algorithmics Group

Kruskal's Algorithm

Karlsruhe Institute of Technology

Use cut and cycle property to merge subtrees of MST

1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)

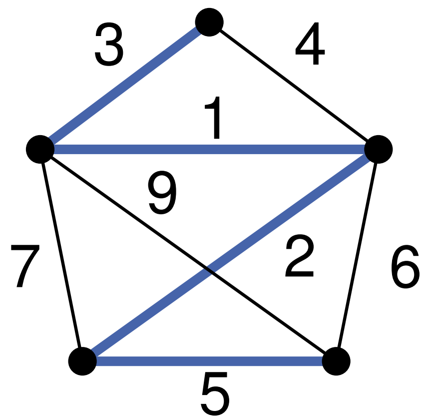


Kruskal's Algorithm

Karlsruhe Institute of Technology

Use cut and cycle property to merge subtrees of MST

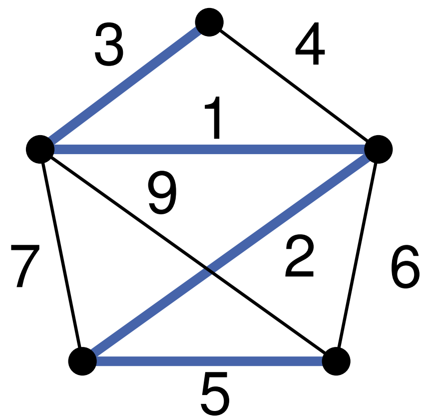
1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)



Kruskal's Algorithm

Use cut and cycle property to merge subtrees of MST

1. Start with empty MST T
2. Sort edges in ascending order of weight
3. Iterate over all edges $\{u, v\}$
   (a) u, v in different subtrees $\Rightarrow T = T \cup \{\{u, v\}\}$ (cut property)
   (b) u, v in same subtree $\Rightarrow$ continue (cycle property)



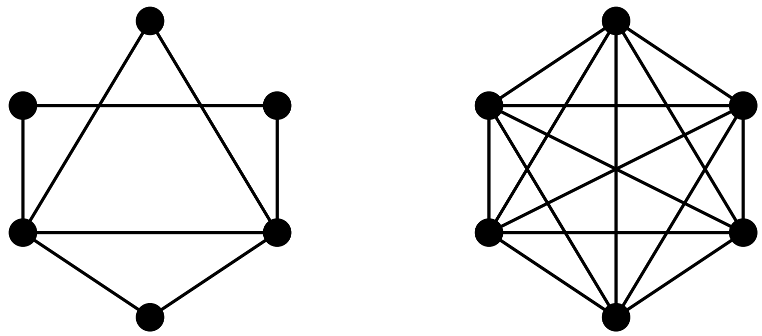
$\Rightarrow$ Fast merging of subtrees using Union-Find

Good $\mathcal{O}(m \log m)$

Comparison

Pro Jarnik-Prim

- Asymptotically good for all m, n
- Very fast for $m \gg n$



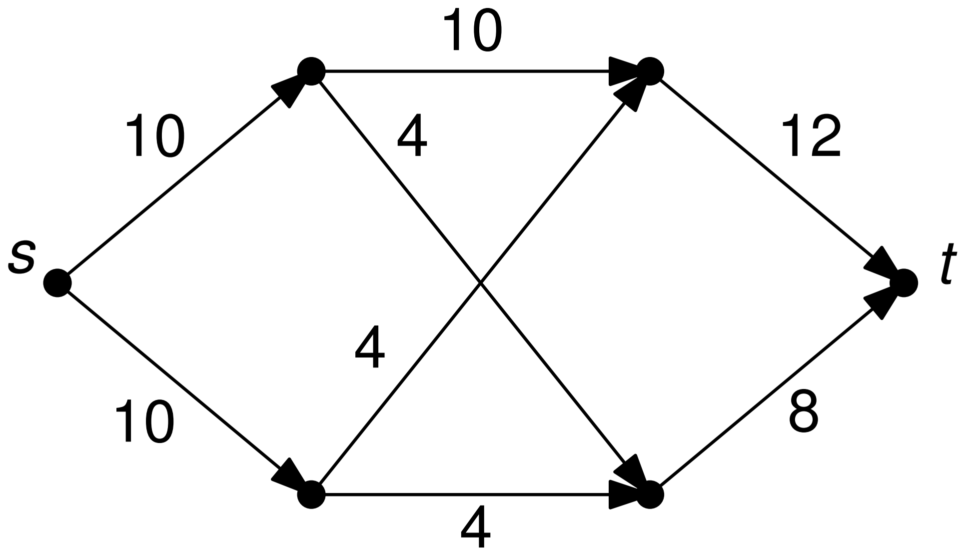
Pro Kruskal

- Fast for $m = \mathcal{O}(n)$
- Only requires adjacency lists
- Profits from fast sorting (e.g. parallel/integers)
- Additional improvements available (e.g. FilterKruskal)

$\Rightarrow$ Choose algorithm based on structure of graph

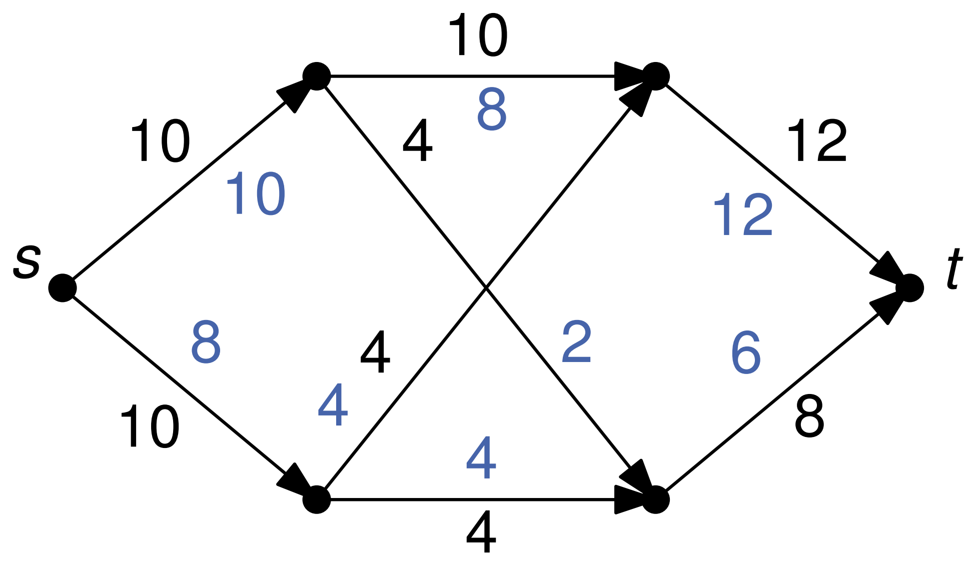
Flow Networks (1/3)

- Network
  - Directed graph $G = (V, E, c)$
  - Source node s ($d_{\text{out}}(s) > 0$)
  - Sink node t ($d_{\text{in}}(t) > 0$)
  - Edge capacity $c(e) > 0$



Flow Networks (2/3)

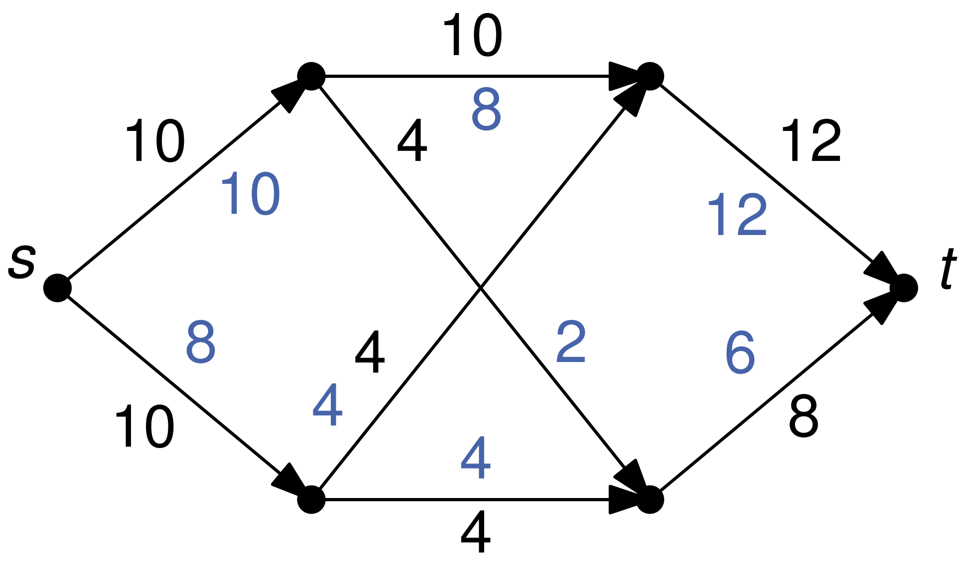
- Flow $f : E \to \mathcal{R}^+$
  - For each edge $e \in E : 0 \leq f(e) \leq c(e)$
  - For each vertex $v \in V \setminus \{s, t\} : \sum_{u \in \Gamma_{\text{in}}} f(u, v) = \sum_{u \in \Gamma_{\text{out}}} f(v, u)$
  - $\text{val}(f) = \sum_{u \in V} f(s, u) - \sum_{u \in V} f(u, s) = \sum_{u \in V} f(u, t) - \sum_{u \in V} f(t, u)$



Institute of Theoretical Informatics
Algorithmics Group

Flow Networks (2/3)

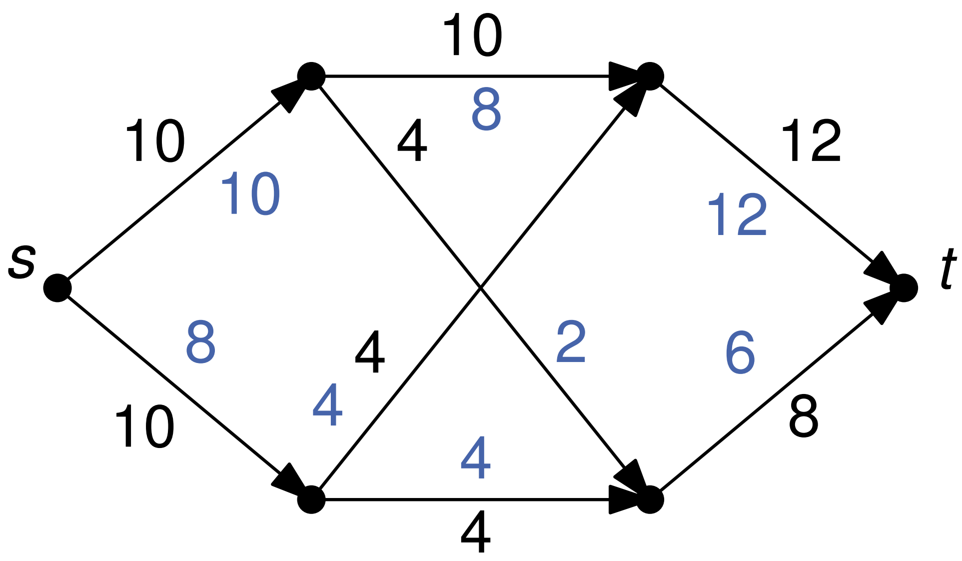
- Flow $f : E \to \mathcal{R}^+$
  - Flow is non-negative and limited by capacity
  - For each vertex $v \in V \setminus \{s, t\} : \sum_{u \in \Gamma_{\text{in}}} f(u, v) = \sum_{u \in \Gamma_{\text{out}}} f(v, u)$
  - $\text{val}(f) = \sum_{u \in V} f(s, u) - \sum_{u \in V} f(u, s) = \sum_{u \in V} f(u, t) - \sum_{u \in V} f(t, u)$



Institute of Theoretical Informatics
Algorithmics Group

Flow Networks (2/3)

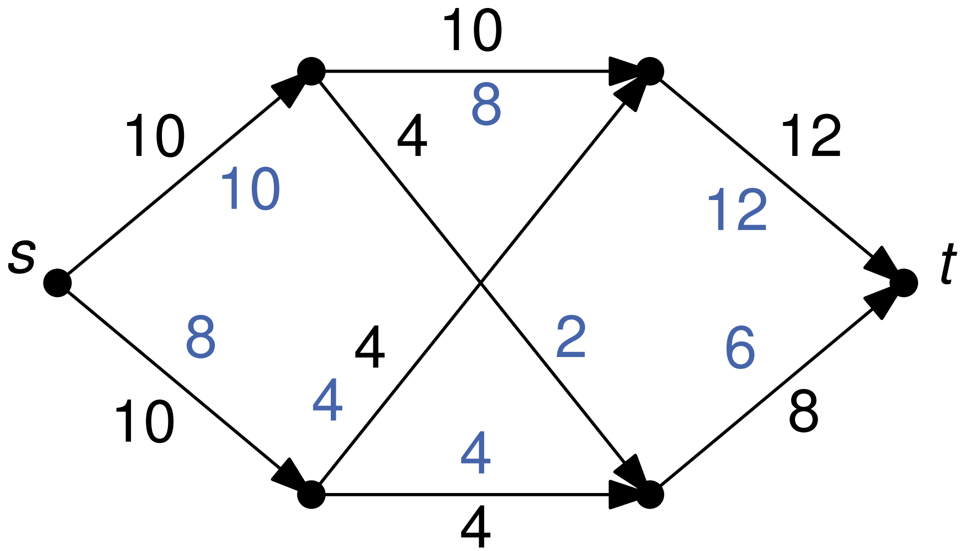
- Flow $f : E \to \mathcal{R}^+$
  - Flow is non-negative and limited by capacity
  - Incoming flow = outgoing flow for each intermediate vertex
  - $\mathrm{val}(f) = \sum_{u \in V} f(s,u) - \sum_{u \in V} f(u,s) = \sum_{u \in V} f(u,t) - \sum_{u \in V} f(t,u)$



Institute of Theoretical Informatics
Algorithmics Group

Flow Networks (2/3)

- Flow $f : E \to \mathcal{R}^+$
  - Flow is non-negative and limited by capacity
  - Incoming flow = outgoing flow for each intermediate vertex
  - Value of flow is outgoing/incoming flow from s/t

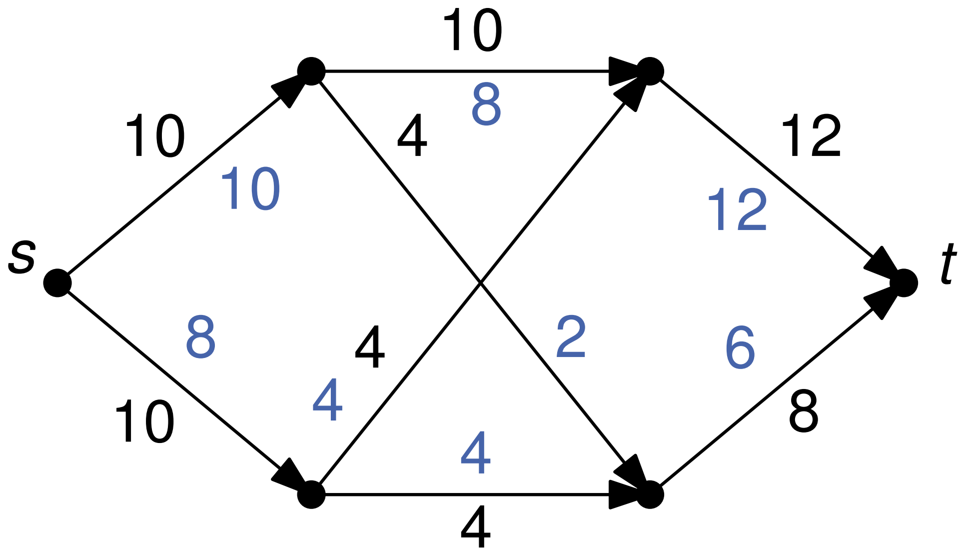


Institute of Theoretical Informatics
Algorithmics Group

Flow Networks (2/3)

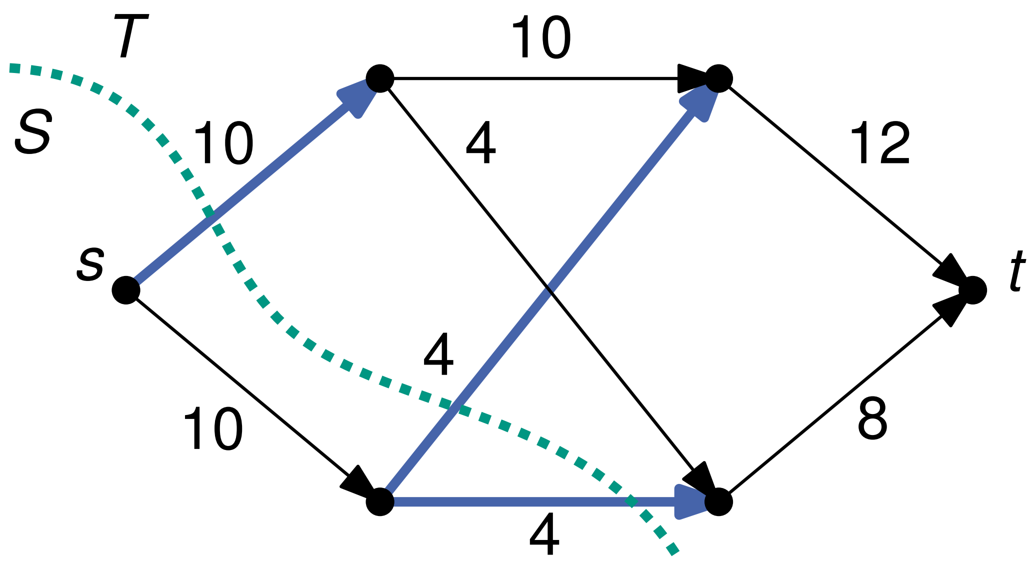
- Flow $f : E \rightarrow \mathcal{R}^+$
  - Flow is non-negative and limited by capacity
  - Incoming flow = outgoing flow for each intermediate vertex
  - Value of flow is outgoing/incoming flow from s/t

$\Rightarrow$ Find flow f with maximum value



Flow Networks (3/3)

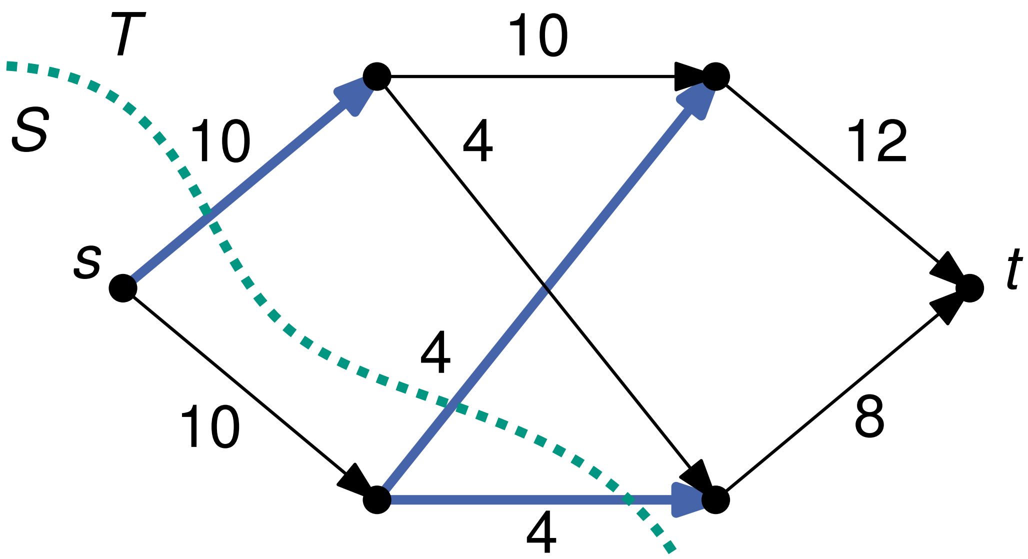
- (Minimum) $s - t$ cuts
  - Partition $V = S \cup T$ into disjoint sets S and T
  - $s \in S$ and $t \in T$
- Capacity of cut is $\sum\{c(u, v) : u \in S, v \in T\}$



Institute of Theoretical Informatics
Algorithmics Group

Flow Networks (3/3)

- (Minimum) $s - t$ cuts
  - Partition $V = S \cup T$ into disjoint sets S and T
  - $s \in S$ and $t \in T$
- Capacity of cut is $\sum\{c(u, v) : u \in S, v \in T\}$



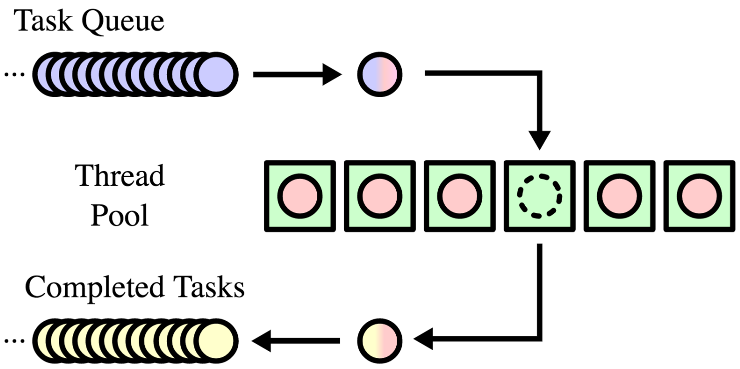
$\Rightarrow$ Duality: Capacity of min. $s - t$ cut = value of max. $s - t$ flow

Applications

Oil pipelines

Traffic flow on highways

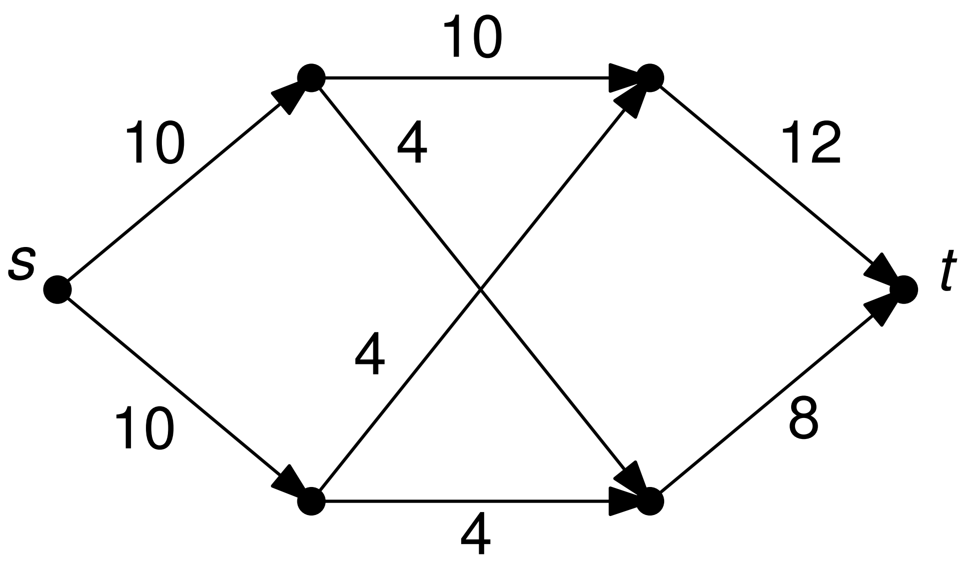
Image processing



Task scheduling

Ford Fulkerson Algorithm

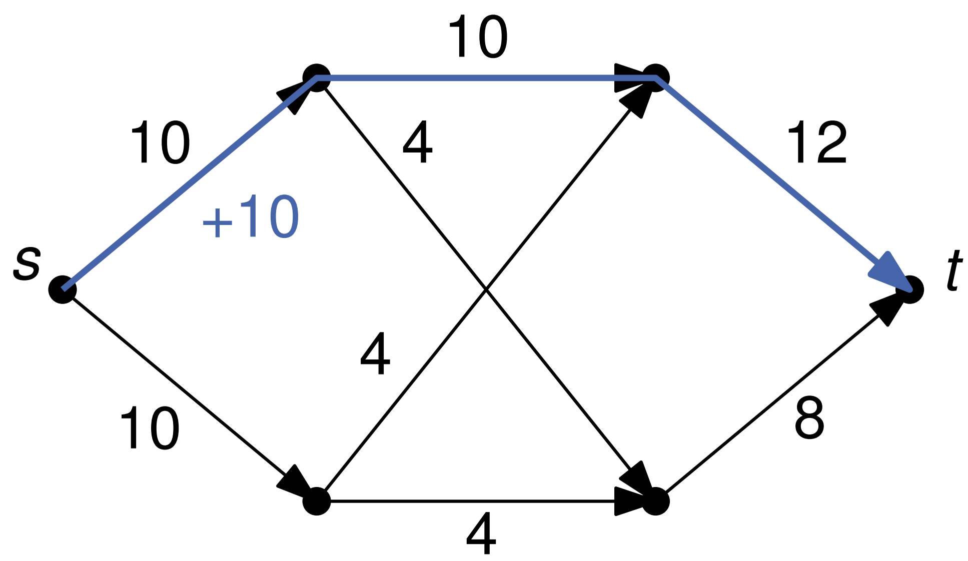
- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



Institute of Theoretical Informatics
Algorithmics Group

Ford Fulkerson Algorithm

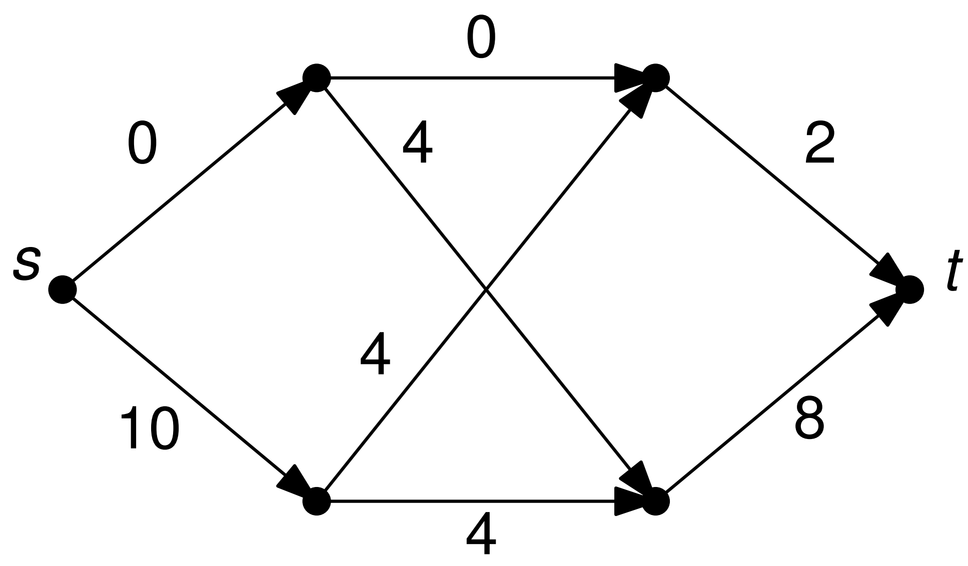
- General Idea (augmenting paths)
  - Find $s-t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



Institute of Theoretical Informatics
Algorithmics Group

Ford Fulkerson Algorithm

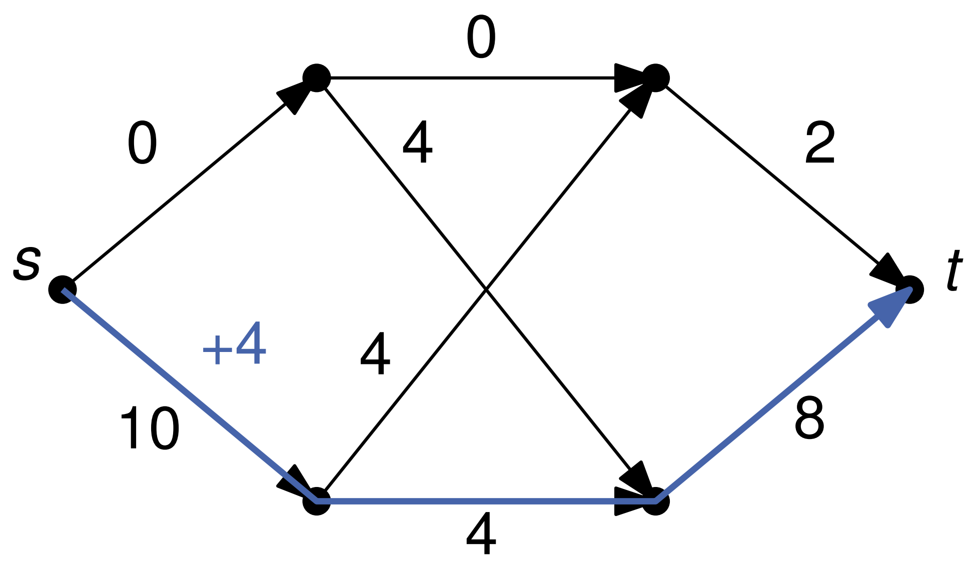
- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



Institute of Theoretical Informatics
Algorithmics Group

Ford Fulkerson Algorithm

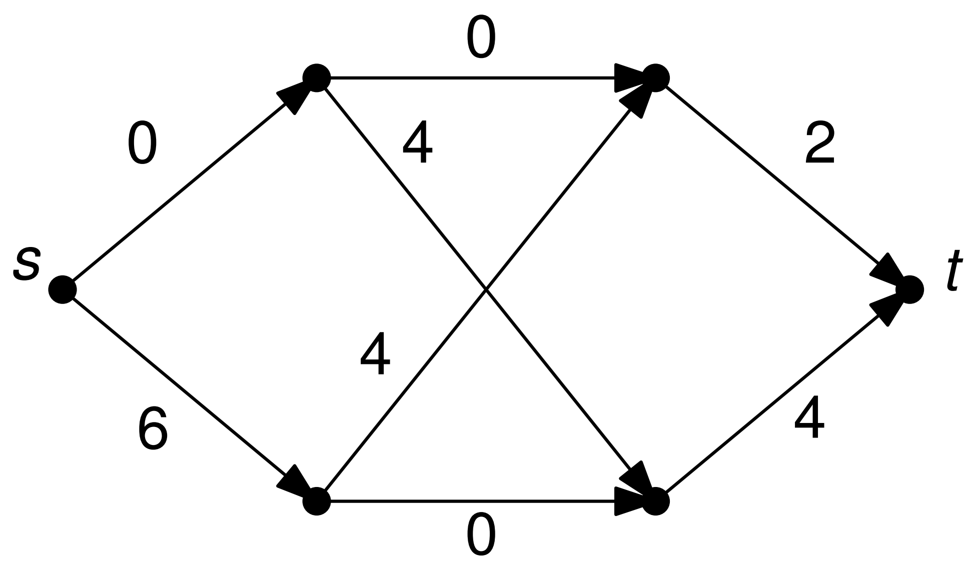
- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



Institute of Theoretical Informatics
Algorithmics Group

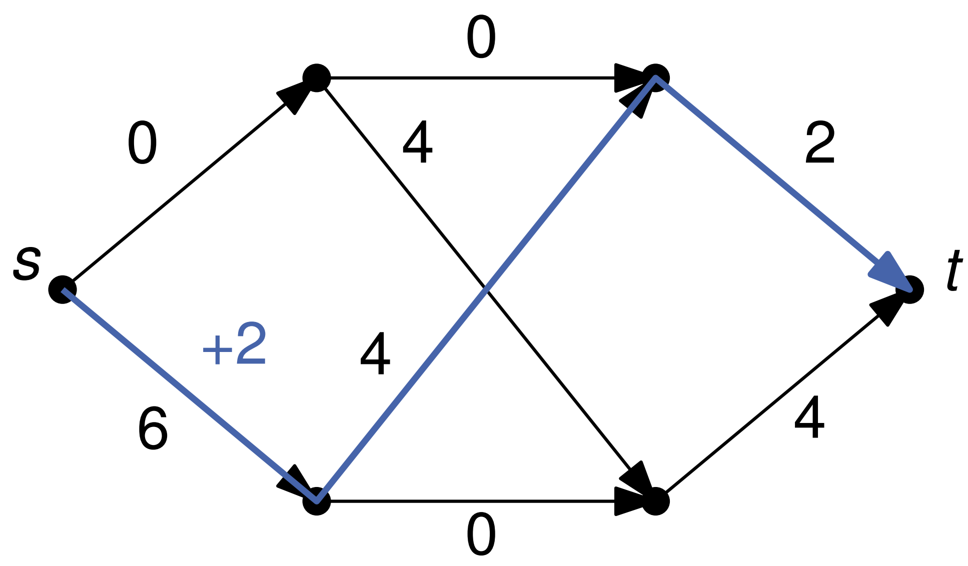
Ford Fulkerson Algorithm

- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



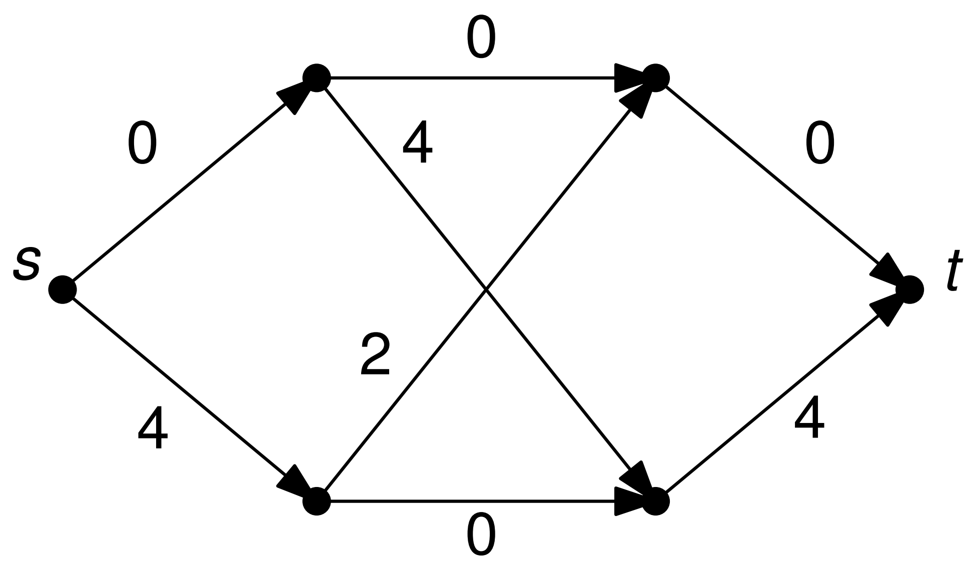
Ford Fulkerson Algorithm

- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



Ford Fulkerson Algorithm

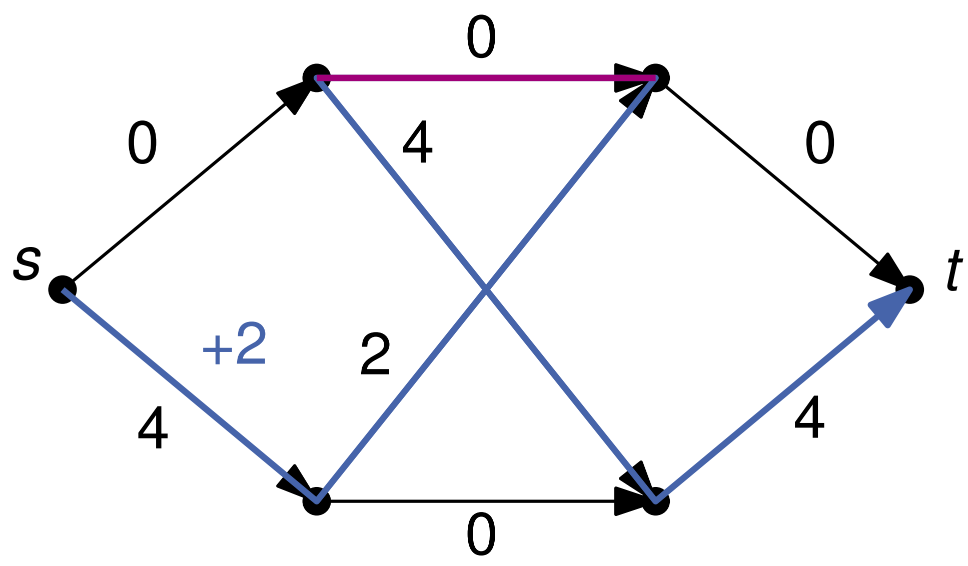
- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



Institute of Theoretical Informatics
Algorithmics Group

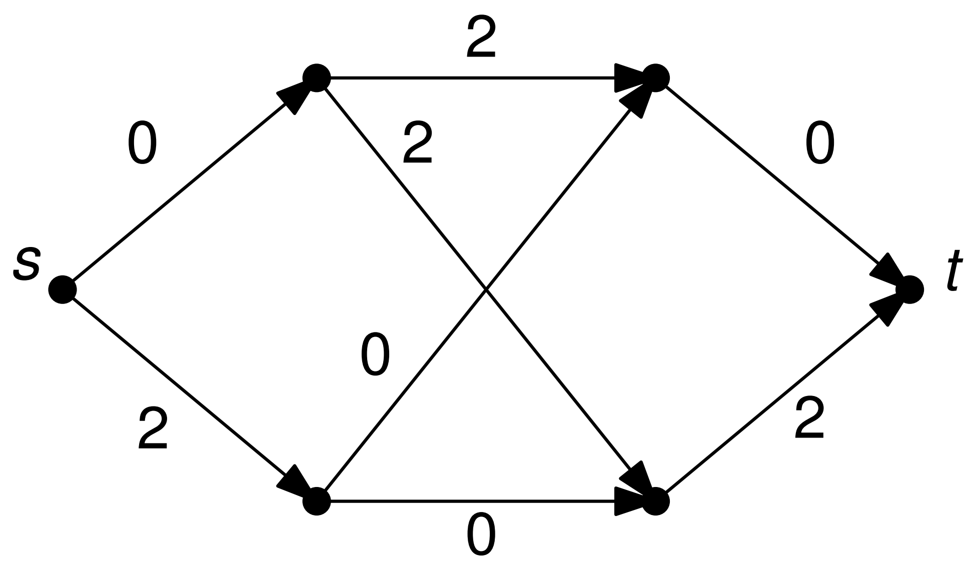
Ford Fulkerson Algorithm

- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



Ford Fulkerson Algorithm

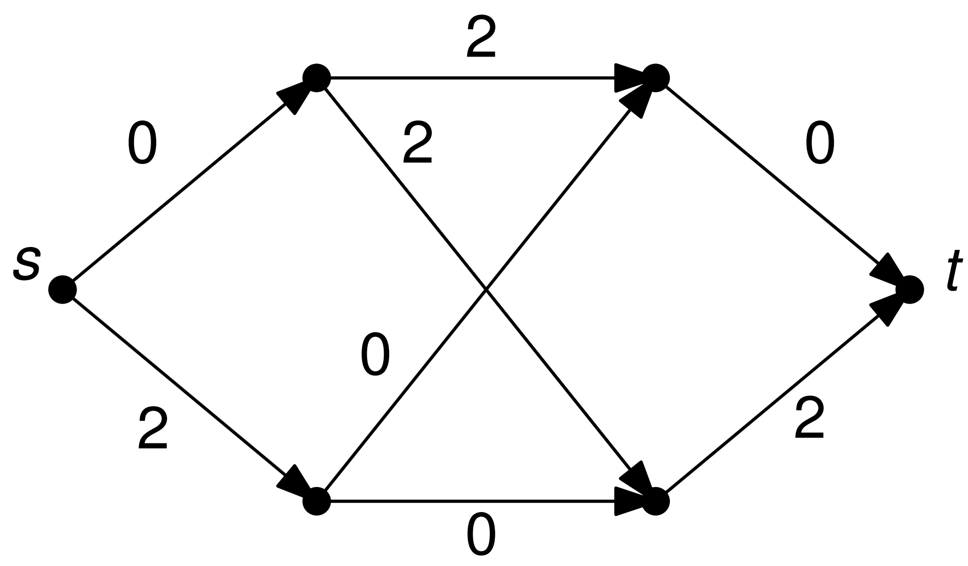
- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)



No more augmenting path

Ford Fulkerson Algorithm

- General Idea (augmenting paths)
  - Find $s - t$ path with spare capacity
  - Sature edge with smallest spare capacity
  - Adjust remaining capacities (create residual graph)

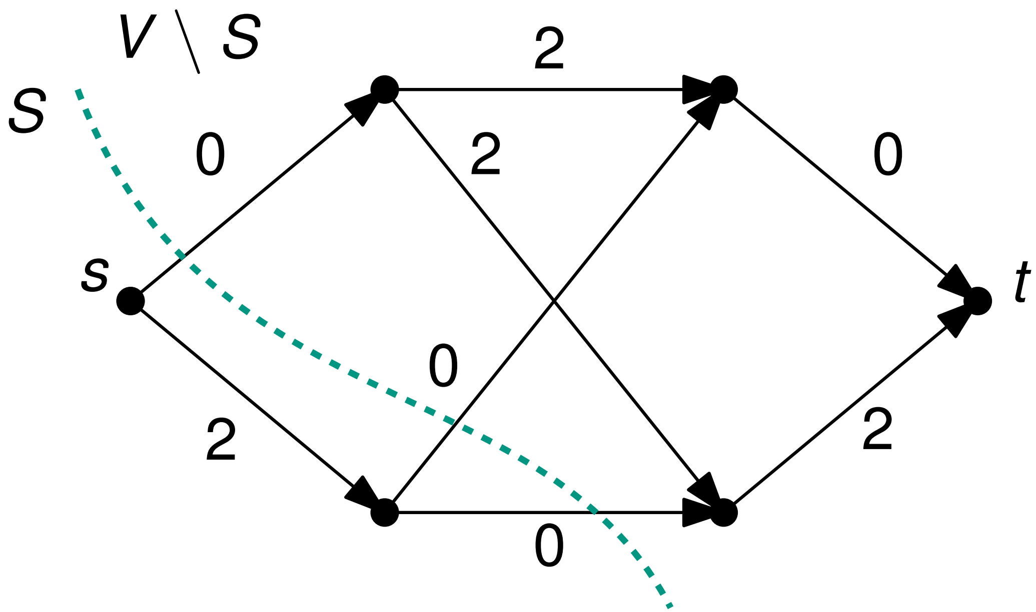


Goodish $\mathcal{O}(m \cdot \text{val}(f))$

Ford Fulkerson Correctness (1/2)

Trivial: Ford Fulkerson computes valid flow
$\Rightarrow$ Remaining: show that flow value is maximal

- At termination we have no augmenting paths in G_f
- Define cut $(S, V \setminus S)$ with $S := \{v \in V : v \text{ reachable from } s \text{ in } G_f\}$



Institute of Theoretical Informatics
Algorithmics Group

Ford Fulkerson Correctness (2/2)

Lemma 1: For any cut (S, T):

$$\mathrm{val}(f) = \underbrace{\boxed{\sum_{e \in E \cap S \times T} f_e}}_{S \to T \text{ edges}} - \underbrace{\boxed{\sum_{e \in E \cap T \times S} f_e}}_{T \to S \text{ edges}}$$

Lemma 2: For each edge $e \in E : c_f(e) = 0 \Rightarrow f(e) = 0$

Institute of Theoretical Informatics
Algorithmics Group

Ford Fulkerson Correctness (2/2)

Lemma 1: For any cut (S, T):

$$\mathrm{val}(f) = \underbrace{\boxed{\sum_{e \in E \cap S \times T} f_e}}_{S \rightarrow T \text{ edges}} - \underbrace{\boxed{\sum_{e \in E \cap T \times S} f_e}}_{T \rightarrow S \text{ edges}}$$

Lemma 2: For each edge $e \in E : c_f(e) = 0 \Rightarrow f(e) = 0$

Observation: For each edge $e \in E \cap S \times T : c_f(e) = 0 \overset{\text{Lemma 2}}{\Rightarrow} f(e) = 0$

$$\begin{aligned} \mathrm{val}(f) &\overset{\text{Lemma 1}}{=} \sum_{e \in E \cap S \times T} f_e - \sum_{e \in E \cap T \times S} f_e \\ &= \sum_{e \in E \cap S \times T} f_e = \text{cut capacity} \\ &\geq \text{maximum flow} \end{aligned}$$

Ford Fulkerson Correctness (2/2)

Lemma 1: For any cut (S, T):

$$\text{val}(f) = \underbrace{\boxed{\sum_{e \in E \cap S \times T} f_e}}_{S \to T \text{ edges}} - \underbrace{\boxed{\sum_{e \in E \cap T \times S} f_e}}_{T \to S \text{ edges}}$$

Lemma 2: For each edge $e \in E : c_f(e) = 0 \Rightarrow f(e) = 0$

Observation: For each edge $e \in E \cap S \times T : c_f(e) = 0 \overset{\text{Lemma 2}}{\Rightarrow} f(e) = 0$

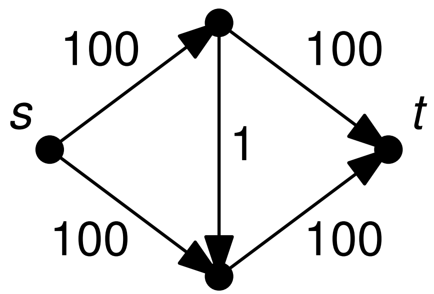
$$\begin{aligned} \text{val}(f) &\overset{\text{Lemma 1}}{=} \sum_{e \in E \cap S \times T} f_e - \sum_{e \in E \cap T \times S} f_e \\ &= \sum_{e \in E \cap S \times T} f_e = \text{cut capacity} \\ &\geq \text{maximum flow} \end{aligned}$$

$\Rightarrow$ Maximum flow = minimum cut

Institute of Theoretical Informatics
Algorithmics Group

Shortcomings of Ford Fulkerson

- Dependence on val(f) can lead to long running times

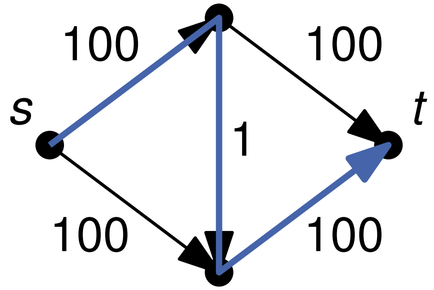


Institute of Theoretical Informatics
Algorithmics Group

Shortcomings of Ford Fulkerson

Karlsruhe Institute of Technology

- Dependence on val(f) can lead to long running times

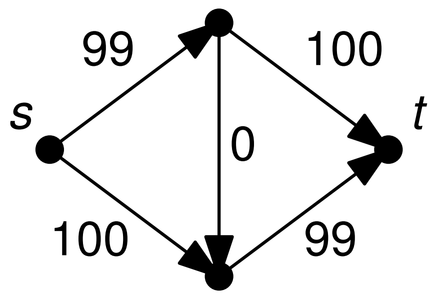


Institute of Theoretical Informatics
Algorithmics Group

Shortcomings of Ford Fulkerson

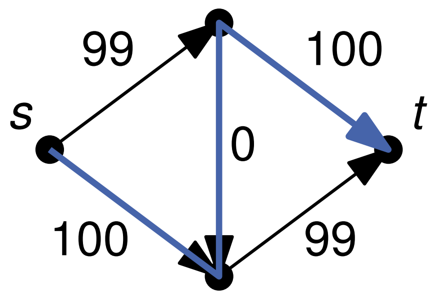
Karlsruhe Institute of Technology

- Dependence on val(f) can lead to long running times



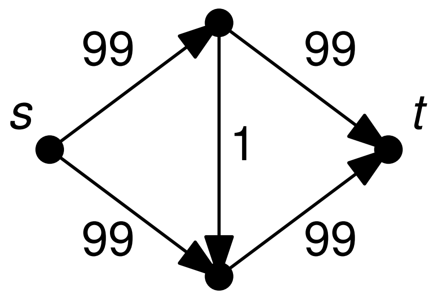
Shortcomings of Ford Fulkerson

- Dependence on val(f) can lead to long running times



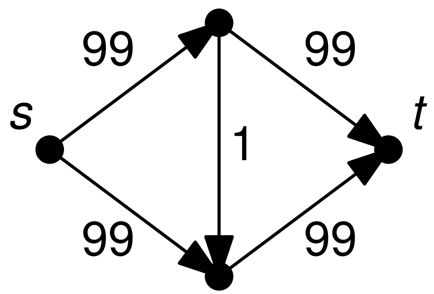
Shortcomings of Ford Fulkerson

- Dependence on val(f) can lead to long running times



Shortcomings of Ford Fulkerson

- Dependence on val(f) can lead to long running times



- Alternatives
  - **1973:** Dinic in $\mathcal{O}(mn \cdot \log(\text{val}(f)))$
  - **1983:** Sleator-Tarjan in $\mathcal{O}(mn \cdot \log(n))$
  - **1986:** Goldberg-Tarjan in $\mathcal{O}(mn \cdot \log(\frac{n^2}{m}))$
  - **1997:** Goldberg-Rao in $\mathcal{O}(\min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\} \cdot m \log(\frac{n^2}{m}) \log U)$
  - **2013:** Orlin and KRT in $\mathcal{O}(mn)$

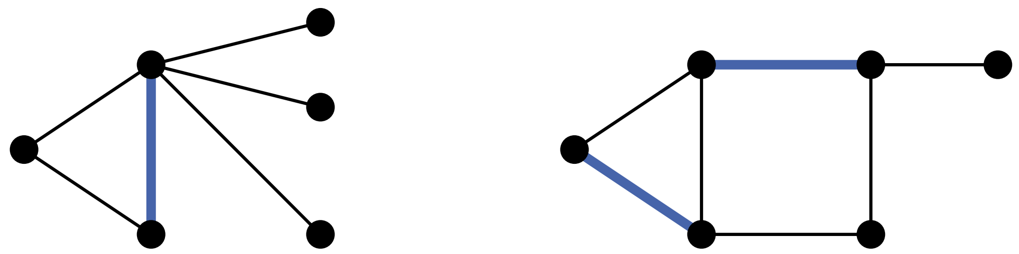
Institute of Theoretical Informatics
Algorithmics Group

Matchings

Given undirected Graph $G = (V, E)$

$M \subseteq E$ is matching $\Leftrightarrow$ M is pairwise non-adjacent

$M \subseteq E$ is maximal matching $\Leftrightarrow$ M is no subset of any other matching in G

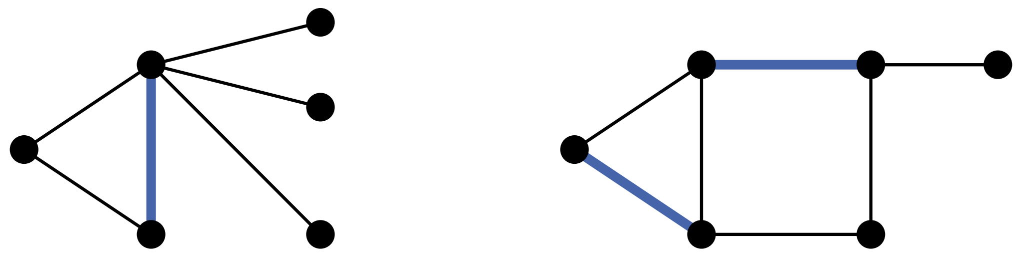


Matchings

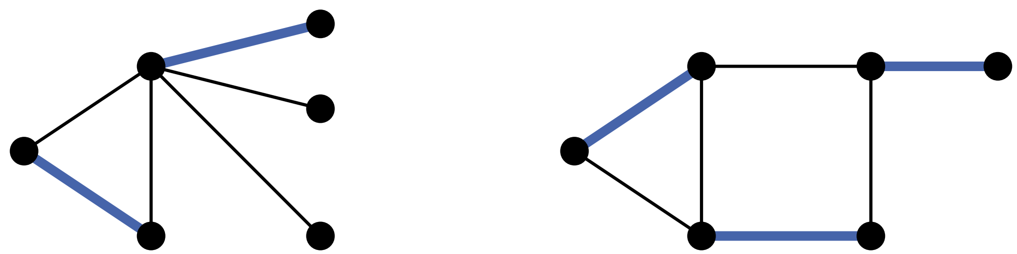
Given undirected Graph $G = (V, E)$

$M \subseteq E$ is matching $\Leftrightarrow$ M is pairwise non-adjacent

$M \subseteq E$ is maximal matching $\Leftrightarrow$ M is no subset of any other matching in G

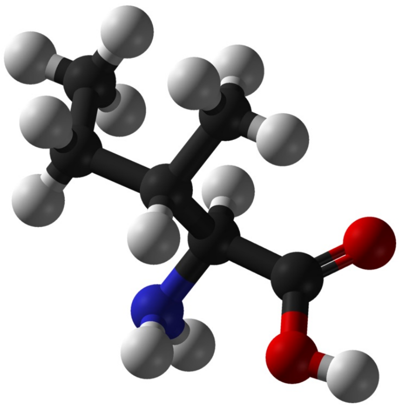


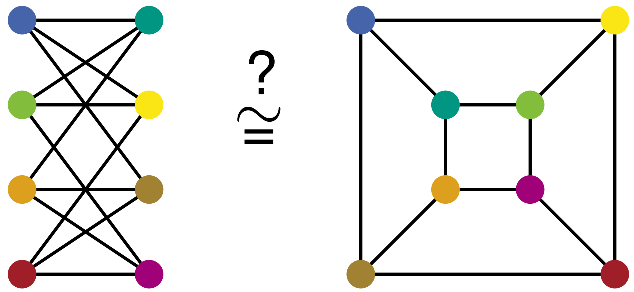
$M \subseteq E$ is maximum matching $\Leftrightarrow$ M has largest possible number of edges



Applications

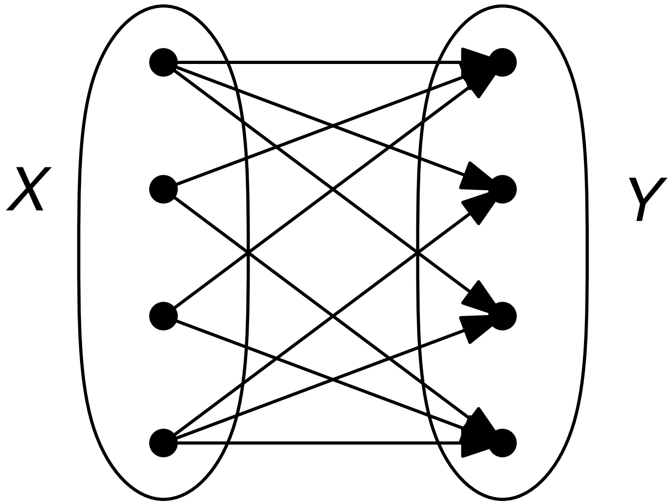
- In general graphs
  - Detection of chemical structures of aromatic compounds
  - Computational/mathematical chemistry (Hosoya index)
- In bipartite graphs
  - Sub-problem for subtree isomorphism
  - Sub-problem for transportation problems





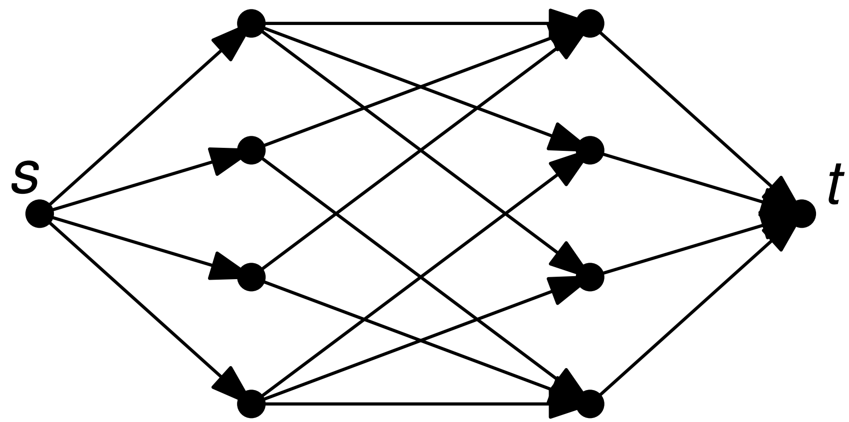
Finding Maximum Bipartite Matchings (1/2)

Given undirected bipartite Graph $G = (V = (X, Y), E)$



Finding Maximum Bipartite Matchings (1/2)

Given undirected bipartite Graph $G = (V = (X, Y), E)$

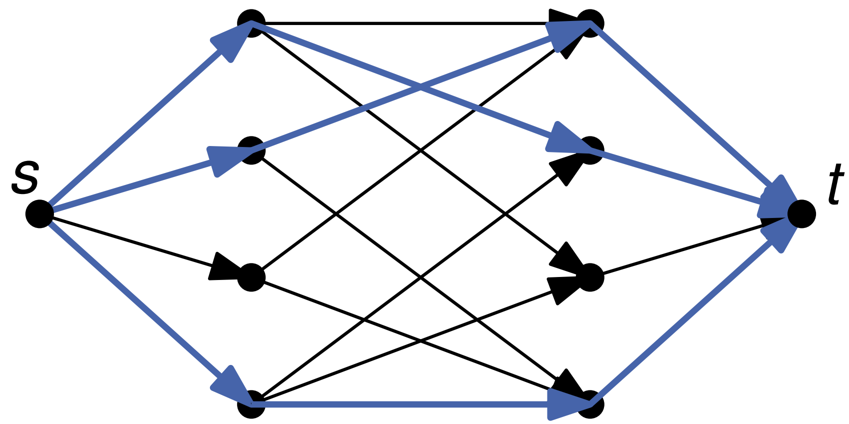


- Algorithm (unit maximum flow)
  1. Direct edges from X to Y
  2. Add super source s and connect to X
  3. Add super sink t and connect to Y

unit costs

Finding Maximum Bipartite Matchings (1/2)

Given undirected bipartite Graph $G = (V = (X, Y), E)$



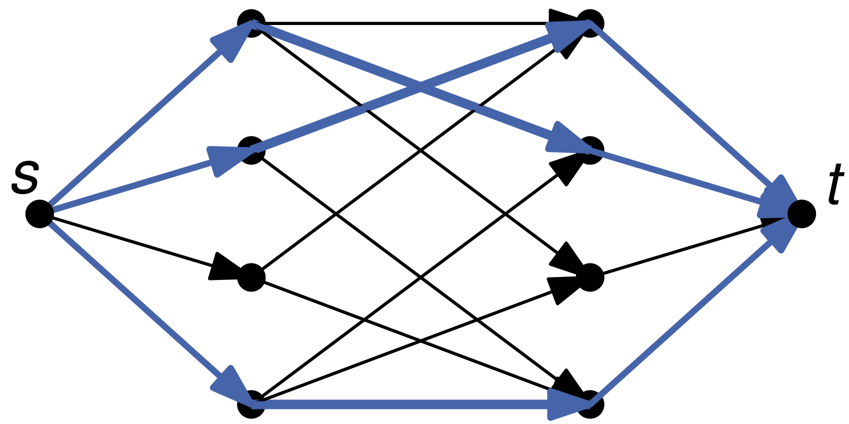
- Algorithm (unit maximum flow)
  1. Direct edges from X to Y
  2. Add super source s and connect to X
  3. Add super sink t and connect to Y

unit costs

$\Rightarrow$ Reduce problem to maximum $s-t$ flow

Finding Maximum Bipartite Matchings (1/2)

Given undirected bipartite Graph $G = (V = (X, Y), E)$



- Algorithm (unit maximum flow)
  1. Direct edges from X to Y
  2. Add super source s and connect to X
  3. Add super sink t and connect to Y

unit costs

$\Rightarrow$ Reduce problem to maximum $s - t$ flow

Goodish $\mathcal{O}(nm)$

Institute of Theoretical Informatics
Algorithmics Group

Finding Maximum Bipartite Matchings (2/2)

Can we do better?

- Hopcroft-Karp in $\mathcal{O}(m\sqrt{n})$
  - Based on augmenting paths
  - Find maximal set of shortest augmenting paths

Institute of Theoretical Informatics
Algorithmics Group

Finding Maximum Bipartite Matchings (2/2)

Can we do better?

- Hopcroft-Karp in $\mathcal{O}(m\sqrt{n})$
  - Based on augmenting paths
  - Find maximal set of shortest augmenting paths
- Madry's algorithm using electric flows in $\mathcal{O}(m^{\frac{10}{7}})$
  - Good for sparse graphs

Institute of Theoretical Informatics
Algorithmics Group

Finding Maximum Bipartite Matchings (2/2)

Can we do better?

- Hopcroft-Karp in $\mathcal{O}(m\sqrt{n})$
  - Based on augmenting paths
  - Find maximal set of shortest augmenting paths
- Madry's algorithm using electric flows in $\mathcal{O}(m^{\frac{10}{7}})$
  - Good for sparse graphs
- Matrix multiplication in $\mathcal{O}(n^{2.376})$
  - Better in theory for dense graphs
  - In practice Hopcroft-Karp still faster

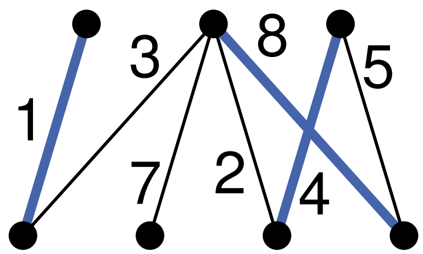
Finding Maximum Bipartite Matchings (2/2)

Can we do better?

- Hopcroft-Karp in $\mathcal{O}(m\sqrt{n})$
  - Based on augmenting paths
  - Find maximal set of shortest augmenting paths
- Madry's algorithm using electric flows in $\mathcal{O}(m^{\frac{10}{7}})$
  - Good for sparse graphs
- Matrix multiplication in $\mathcal{O}(n^{2.376})$
  - Better in theory for dense graphs
  - In practice Hopcroft-Karp still faster
- Chandran and Hochbaum in $\mathcal{O}(\min\{|X|k, m\} + \sqrt{k}\min\{k^2, m\})$
  - Output-sensitive algorithm

Finding Maximum Matchings

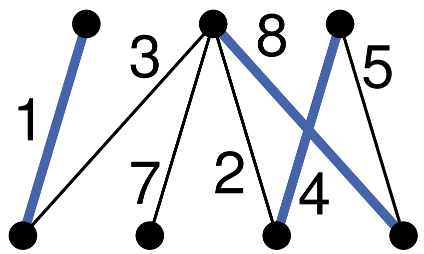
- In weighted bipartite graphs
  - Find matching with maximum value
  - Modified augmenting paths algorithm in $\mathcal{O}(n^2 \log n + nm)$



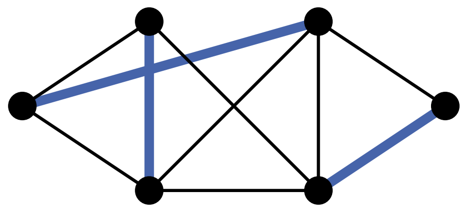
Institute of Theoretical Informatics
Algorithmics Group

Finding Maximum Matchings

- In weighted bipartite graphs
  - Find matching with maximum value
  - Modified augmenting paths algorithm in $\mathcal{O}(n^2 \log n + nm)$



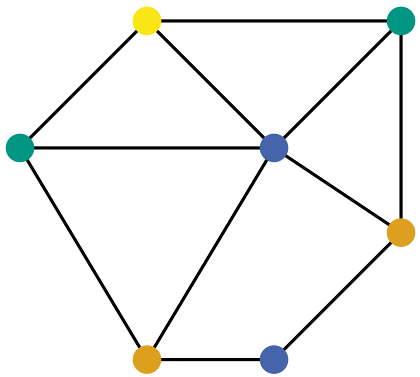
- In general graphs
  - Edmonds' algorithm in $\mathcal{O}(n^2 m)$
  - Improved version in time $\mathcal{O}(\sqrt{n}m)$

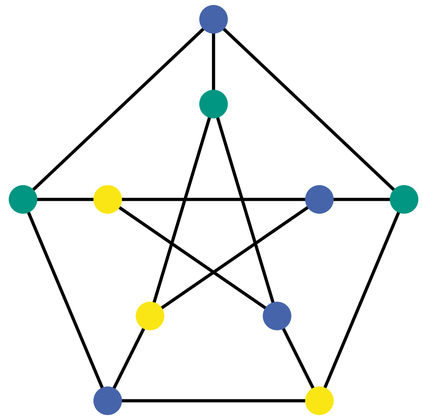


Coloring

Given undirected Graph $G = (V, E)$ (without self-loops)

- Vertex coloring
  - Label each vertex with a color
  - No two vertices sharing an edge have the same color

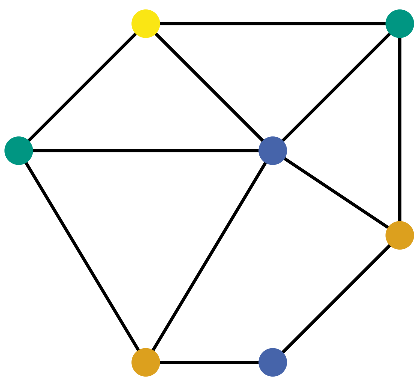




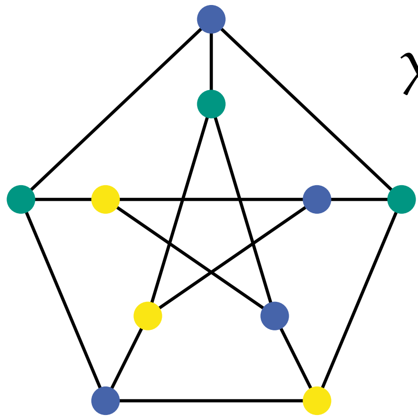
Coloring

Given undirected Graph $G = (V, E)$ (without self-loops)

- Vertex coloring
  - Label each vertex with a color
  - No two vertices sharing an edge have the same color



$\chi(G) = 4$



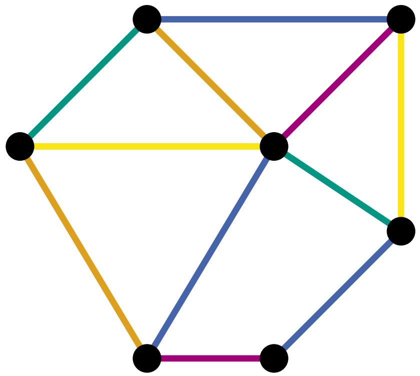
$\chi(G) = 3$

- k-coloring
  - Vertex coloring that uses at most k-colors
  - Smallest possible k of G is called chromatic number $\chi(G)$

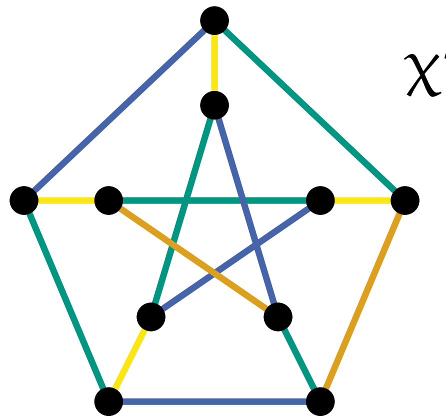
Institute of Theoretical Informatics
Algorithmics Group

Related Problems

- Edge coloring
  - Label each edge with a color
  - No two edges sharing a vertex have the same color



$\chi'(G) = 5$



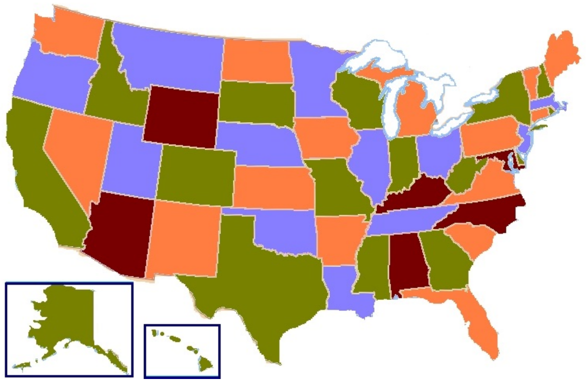
$\chi'(G) = 4$

- Improper colorings (i.e. Ramsey theory)
  - Label each edge with a color
  - Two edges sharing a vertex are allowed the same color
  - Example: Friendship theorem

Applications

Task/Exam scheduling

Sudoku solving



Map coloring

Mobile Radio Frequency Assignment

Finding k-Colorings

- Find vertex coloring with minimum number of colors

  $\Rightarrow$ Optimization problem is NP-hard

Institute of Theoretical Informatics
Algorithmics Group

Finding k-Colorings

- Find vertex coloring with minimum number of colors
  $\Rightarrow$ Optimization problem is NP-hard

- Exact algorithms for general graphs
  - Brute-force search for a k-coloring in $\mathcal{O}(k^n)$
  - Best exact algorithm for finding k-coloring in $\mathcal{O}(2^n n)$

Finding k-Colorings

- Find vertex coloring with minimum number of colors

  $\Rightarrow$ Optimization problem is NP-hard

- Exact algorithms for general graphs
  - Brute-force search for a k-coloring in $\mathcal{O}(k^n)$
  - Best exact algorithm for finding k-coloring in $\mathcal{O}(2^n n)$

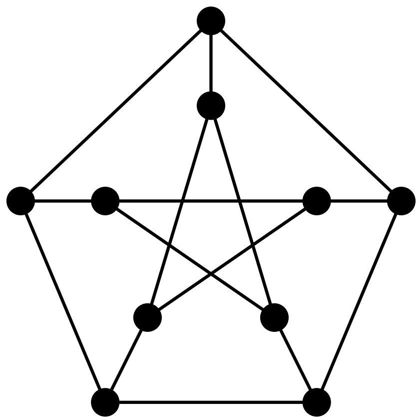
- Even worse for general graphs
  - No constant factor approximations in polynomial time
  - Approximable with absolute error guarantee of 1 on planar graphs

$\Rightarrow$ **Ugly** How to find good heuristics?

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

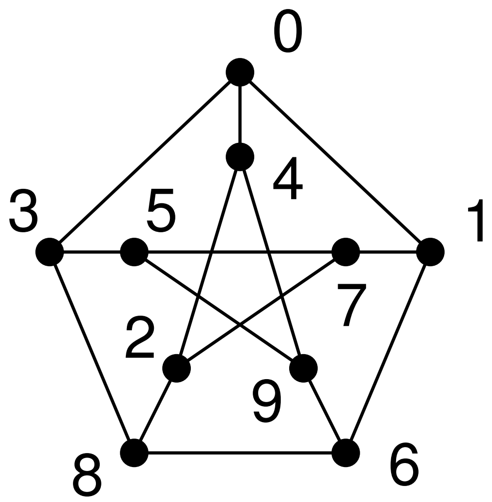
1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

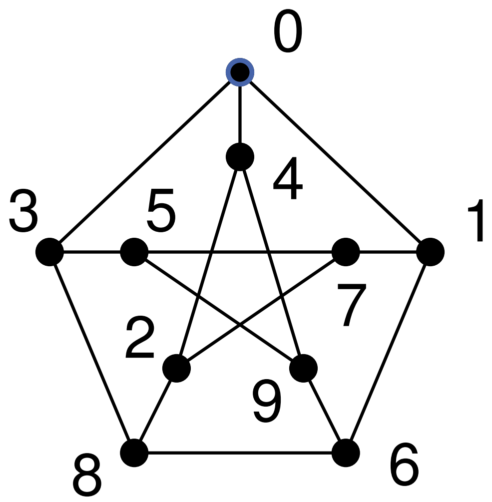
1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

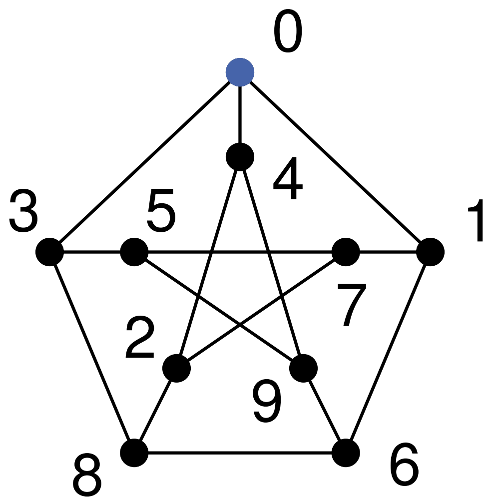


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

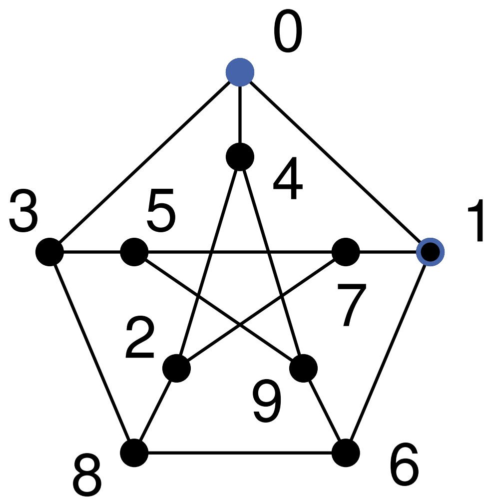
1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

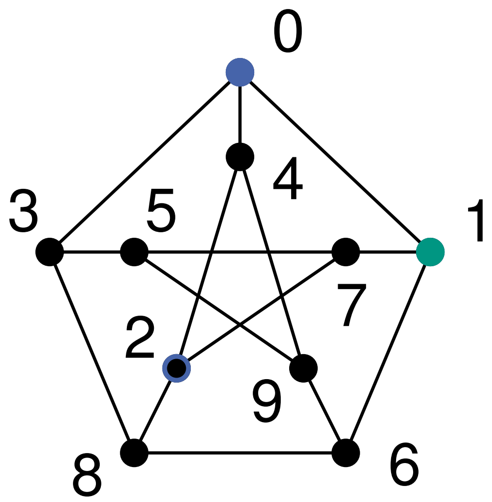


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

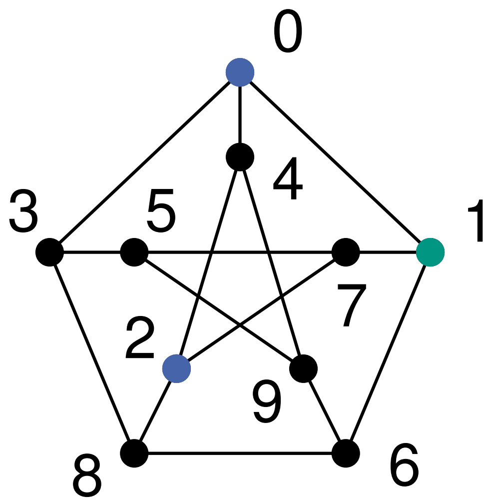


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

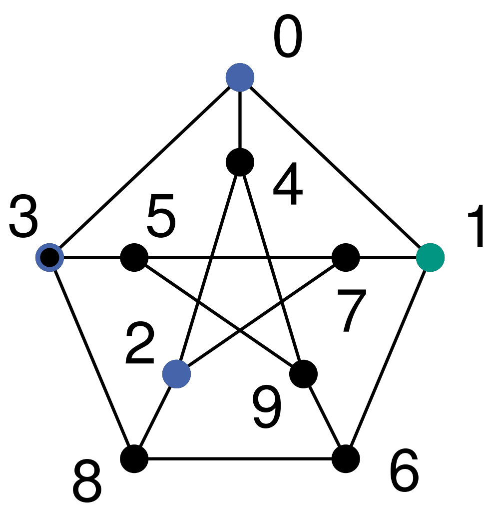


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

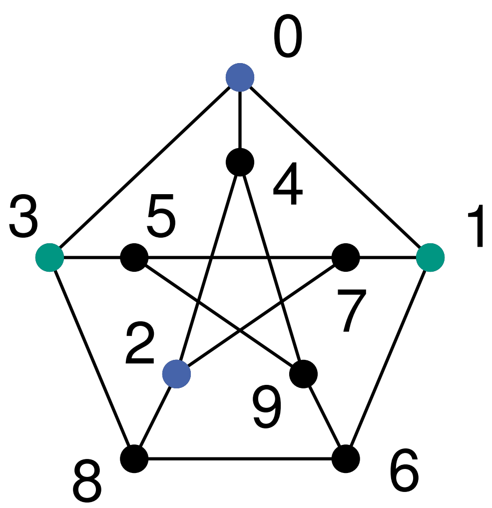


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

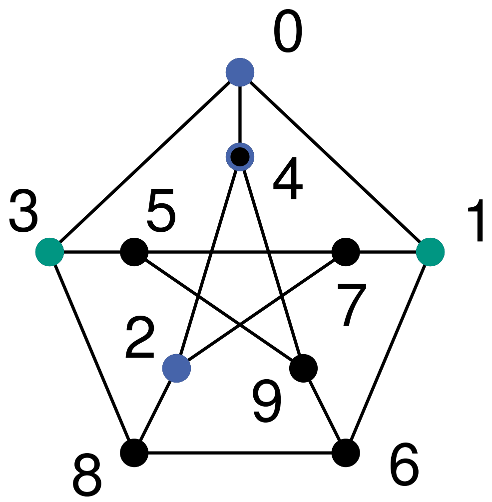


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

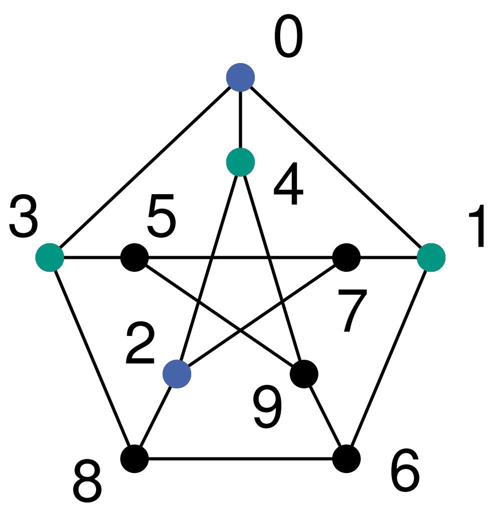


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

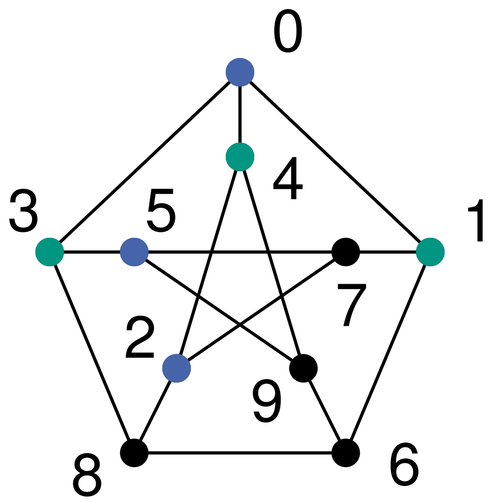


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

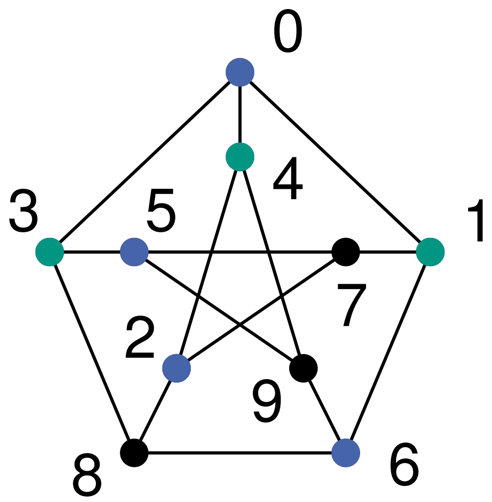
1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

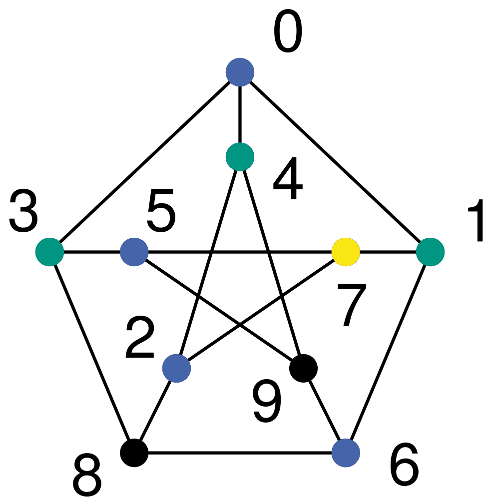
1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

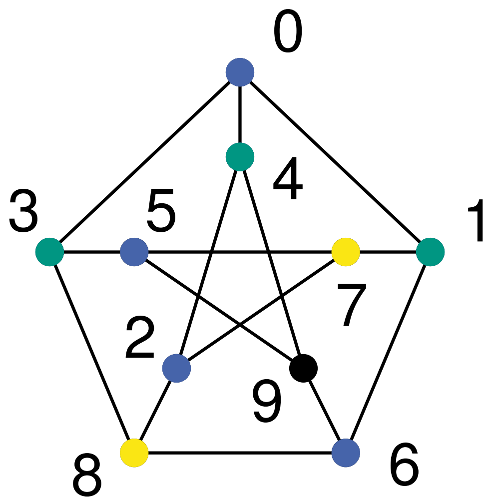


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

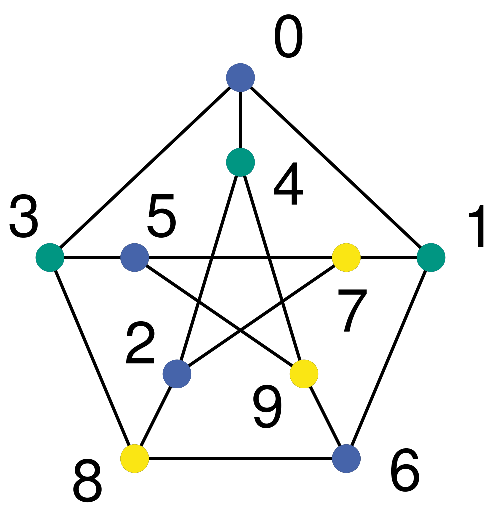
1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

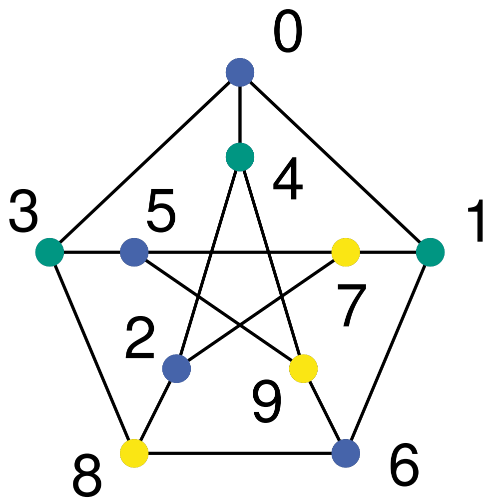


Institute of Theoretical Informatics
Algorithmics Group

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort colors
2. Sort vertices with predefined order
3. Iterate over vertices in sorted order
   (a) Color vertex with smallest color not used by any neighbor
   (b) Add new color if necessary

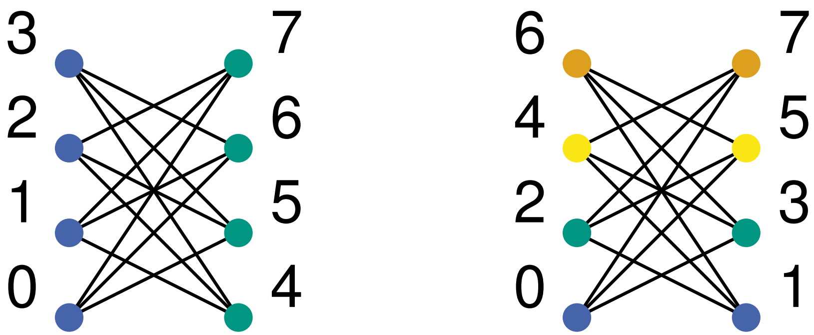


$\Rightarrow$ At most $\Delta + 1$ colors

Good $\mathcal{O}(n + m)$

Shortcomings of Greedy Algorithm

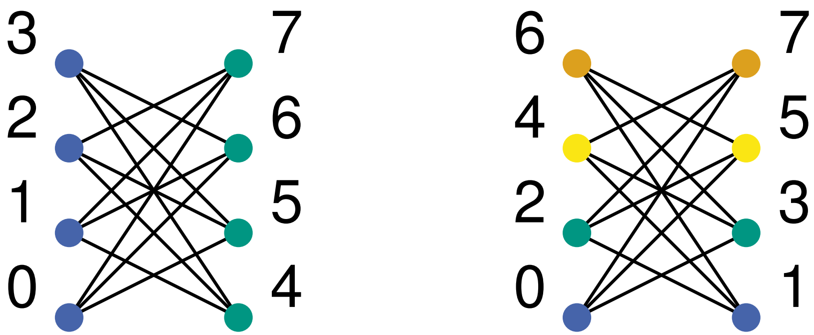
- Quality of approximation heavily dependent on vertex ordering



$\Rightarrow$ Finding perfect ordering is NP-hard

Shortcomings of Greedy Algorithm

- Quality of approximation heavily dependent on vertex ordering

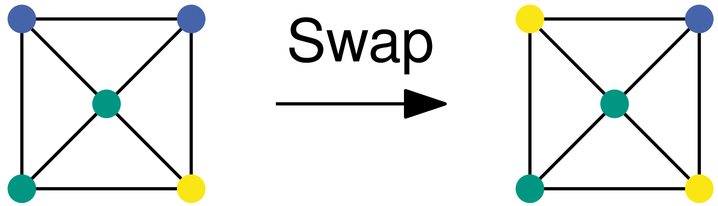


$\Rightarrow$ Finding perfect ordering is NP-hard

- Heuristic ordering strategies
  - Sort orders by their decreasing degree
  - Better upper bound than random ordering

Finding Colorings in Practice

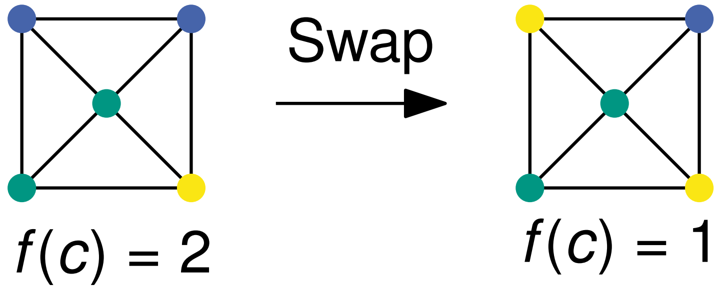
- Tabu search
  - Temporarily allow invalid solutions
  - Minimize conflicts and discourage repetition



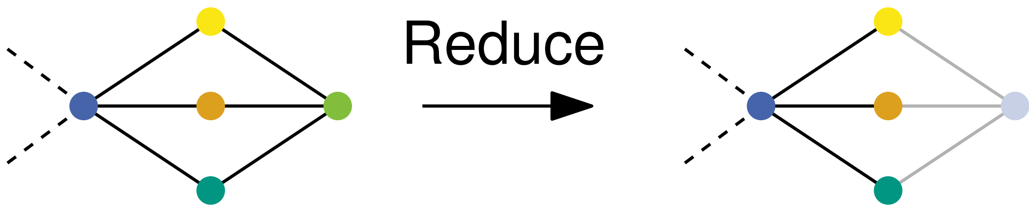
$f(c) = 2$ $\qquad$ $f(c) = 1$

Finding Colorings in Practice

- Tabu search
  - Temporarily allow invalid solutions
  - Minimize conflicts and discourage repetition



- Reductions
  - Remove subgraphs with certain structure
  - Subgraphs can be solved exactly



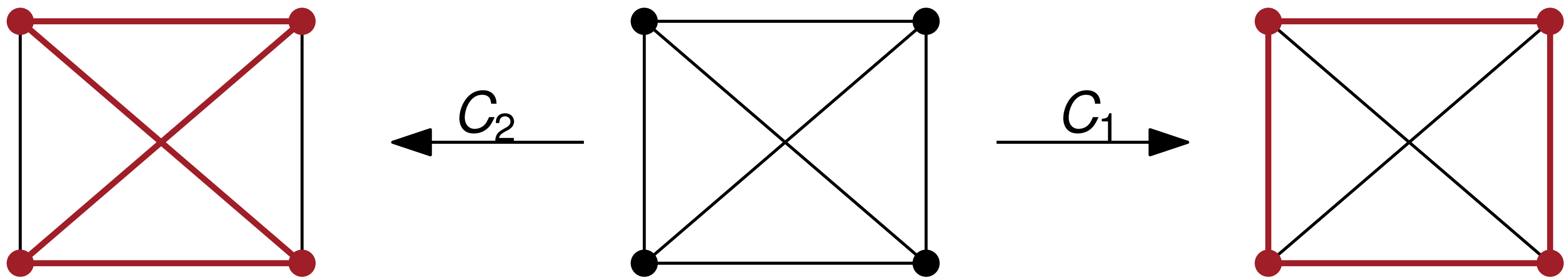
Institute of Theoretical Informatics
Algorithmics Group

Traveling Salesman Problem

TSP is the prototypical optimization problem

Preliminary: Hamiltonian Cycle Problem
Is there a cycle in graph G that visits each vertex exactly once?

$$\mathbb{M} := \{G = (V, E) : \exists C \subseteq E : |C| = |V|, \text{C is a cycle}\}$$

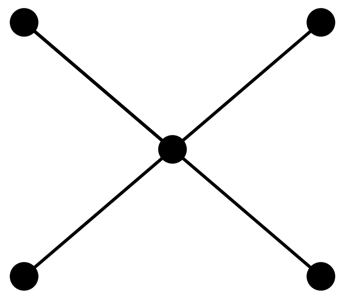


Traveling Salesman Problem

TSP is the prototypical optimization problem

Preliminary: Hamiltonian Cycle Problem
Is there a cycle in graph G that visits each vertex exactly once?

$$\mathbb{M} := \{G = (V, E) : \exists C \subseteq E : |C| = |V|, \text{C is a cycle}\}$$



Institute of Theoretical Informatics
Algorithmics Group

Traveling Salesman Problem

TSP is the prototypical optimization problem

Definition:

Given graph $G = (V, E, \omega)$ find a simple cycle C such that $|C| = |V|$ and

$$\sum_{e \in C} \omega(e) \text{ is minimized.}$$

Institute of Theoretical Informatics
Algorithmics Group

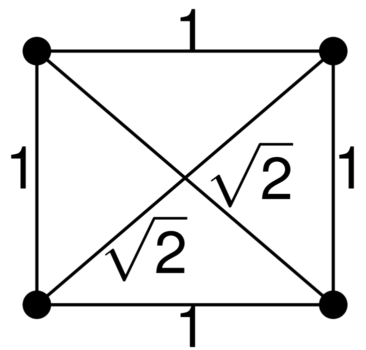
Traveling Salesman Problem

TSP is the prototypical optimization problem

Definition:

Given graph $G = (V, E, \omega)$ find a simple cycle C such that $|C| = |V|$ and

$$\sum_{e \in C} \omega(e) \text{ is minimized.}$$



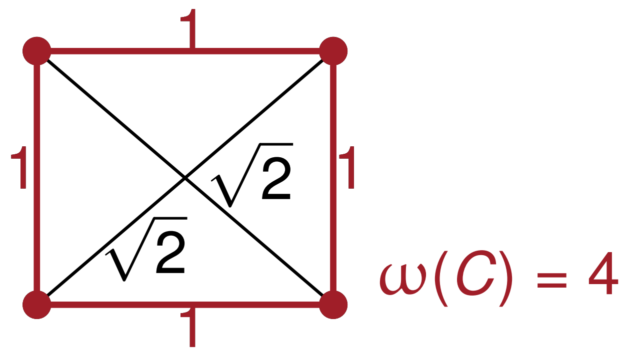
Traveling Salesman Problem

TSP is the prototypical optimization problem

Definition:

Given graph $G = (V, E, \omega)$ find a simple cycle C such that $|C| = |V|$ and

$$\sum_{e \in C} \omega(e) \text{ is minimized.}$$



Institute of Theoretical Informatics
Algorithmics Group

Traveling Salesman Problem

TSP is the prototypical optimization problem

Definition:

Given graph $G = (V, E, \omega)$ find a simple cycle C such that $|C| = |V|$ and

$$\sum_{e \in C} \omega(e) \text{ is minimized.}$$

- the TSP is $\mathbf{NP}$-hard
  If $\omega(e) = c$ for all $e \in E$ then TSP $\sim$ Hamiltonian Cycle
- it is $\mathbf{NP}$-hard to approximate the general TSP within any factor α

Ugly: $\mathbf{NP}$-hard
not $\mathbf{APX}$

Traveling Salesman Problem

It is $\mathbf{NP}$-hard to approximate the general TSP within any factor α.

Given HC instance $G = (V, E)$ consider TSP instance $G' = (V, V \times V)$ and

$$\omega(e) = \begin{cases} 1 & \text{if } e \in E \\ \alpha n & \text{else} \end{cases}$$

- if G has HC $\Leftrightarrow$ there is a TSP tour of weight n in G'
  $\Rightarrow$ α-approx. algorithm delivers tour with weight $\leq \alpha n$
- if G has no HC $\Leftrightarrow$ every TSP tour in G' has weight $\geq \alpha n + n - 1 > \alpha n$
- if α-approx algorithm finds tour with weight $\leq \alpha n$ in G'
  $\Rightarrow$ HC exists in G

Traveling Salesman Problem

It is $\mathbf{NP}$-hard to approximate the general TSP within any factor α.

Given HC instance $G = (V, E)$ consider TSP instance $G' = (V, V \times V)$ and

$$\omega(e) = \begin{cases} 1 & \text{if } e \in E \\ \alpha n & \text{else} \end{cases}$$

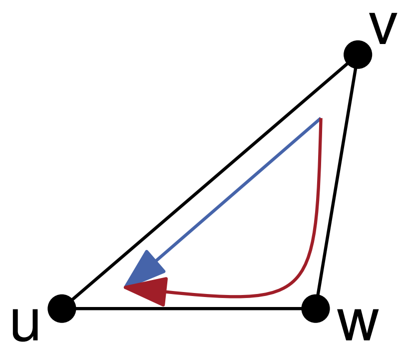
- if G has HC $\Leftrightarrow$ there is a TSP tour of weight n in G'
  $\Rightarrow$ α-approx. algorithm delivers tour with weight $\leq \alpha n$
- if G has no HC $\Leftrightarrow$ every TSP tour in G' has weight $\geq \alpha n + n - 1 > \alpha n$
- if α-approx algorithm finds tour with weight $\leq \alpha n$ in G'
  $\Rightarrow$ HC exists in G

If we restrict the general TSP we can do better

Metric Traveling Salesman Problem

- $G = (V, E, \omega)$ is undirected, connected and obeys the triangle inequality

$$\forall u, v, w \in V : \omega((u, w)) \leq \omega((u, v)) + \omega((v, w))$$



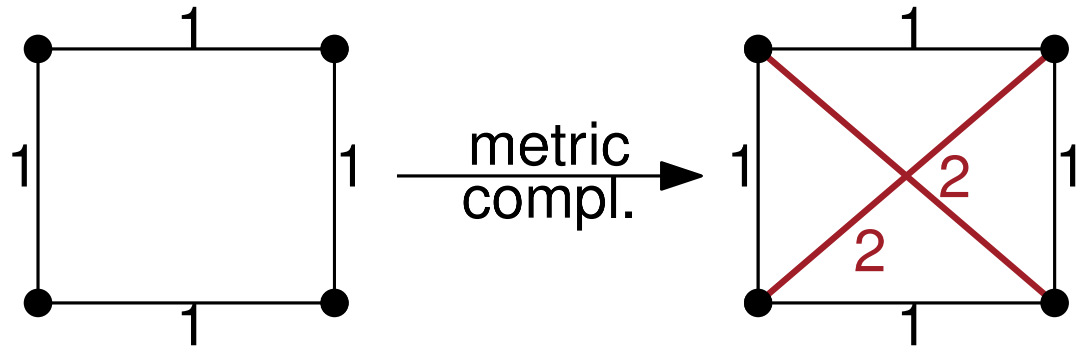
Metric Traveling Salesman Problem

- $G = (V, E, \omega)$ is undirected, connected and obeys the triangle inequality

$$\forall u, v, w \in V : \omega((u, w)) \leq \omega((u, v)) + \omega((v, w))$$

- the metric completion of $G = (V, E, \omega)$ is defined as $G' = (V, V \times V, \omega')$ with

$$\omega'(e = (u, v)) = \begin{cases} \omega(e) & \text{if } e \in E \\ \omega(u, \dots, v) & \text{for shortest path from } u \text{ to } v \text{ in } E \end{cases}$$



Institute of Theoretical Informatics
Algorithmics Group

Metric Traveling Salesman Problem

2-Approximation via MST

Lemma
Given $G = (V, E, \omega)$ and its MST T,

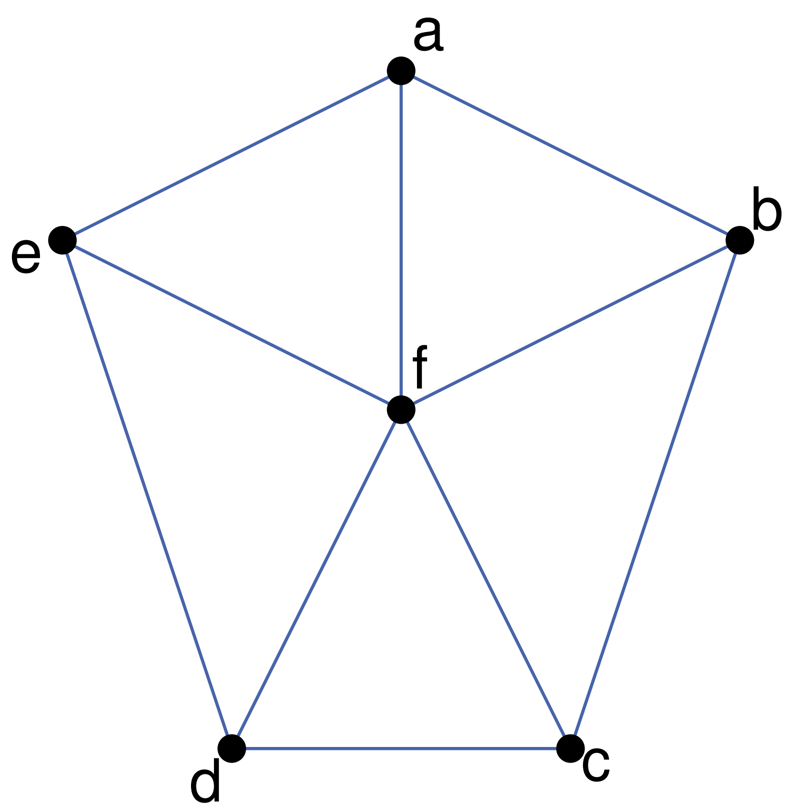
$$\omega(T) \leq \text{weight of any TSP tour of } G.$$

This includes optimal mimimum weight tour OPT.

Metric Traveling Salesman Problem

2-Approximation via MST

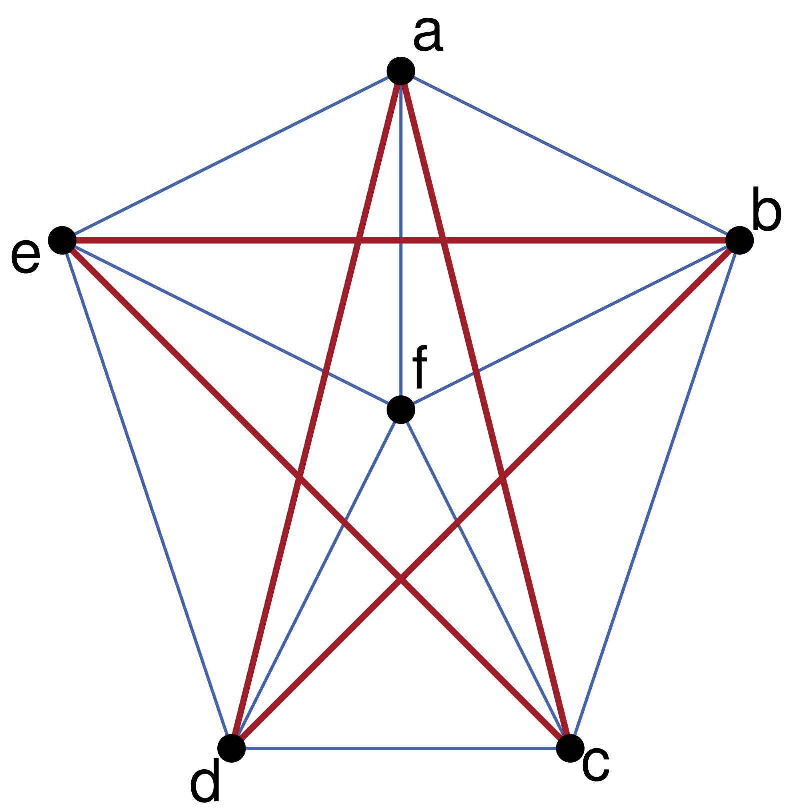
- given $G = (V, E, \omega)$, $\omega(e) = 1$



Metric Traveling Salesman Problem

2-Approximation via MST

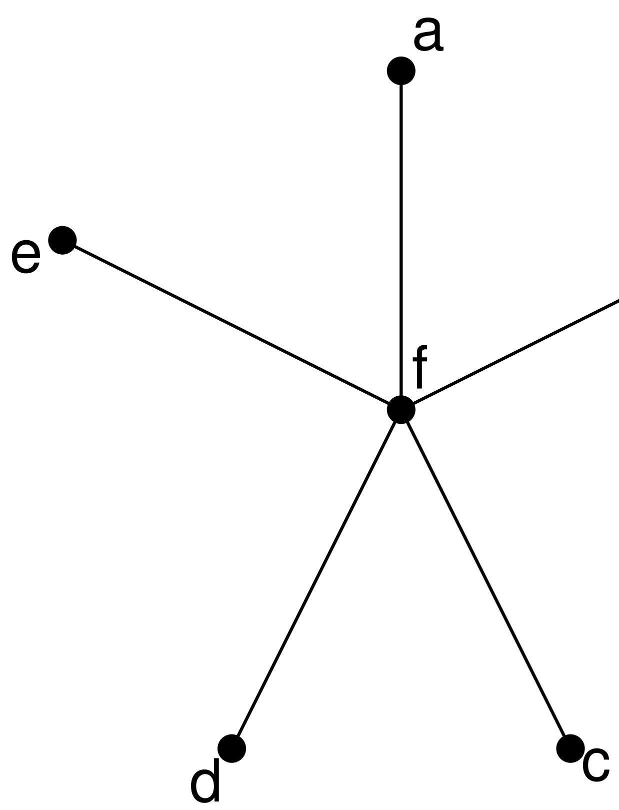
- given $G = (V, E, \omega)$, $\omega(e) = 1$
- metric completion, $\omega(e') = 2$



Metric Traveling Salesman Problem

2-Approximation via MST

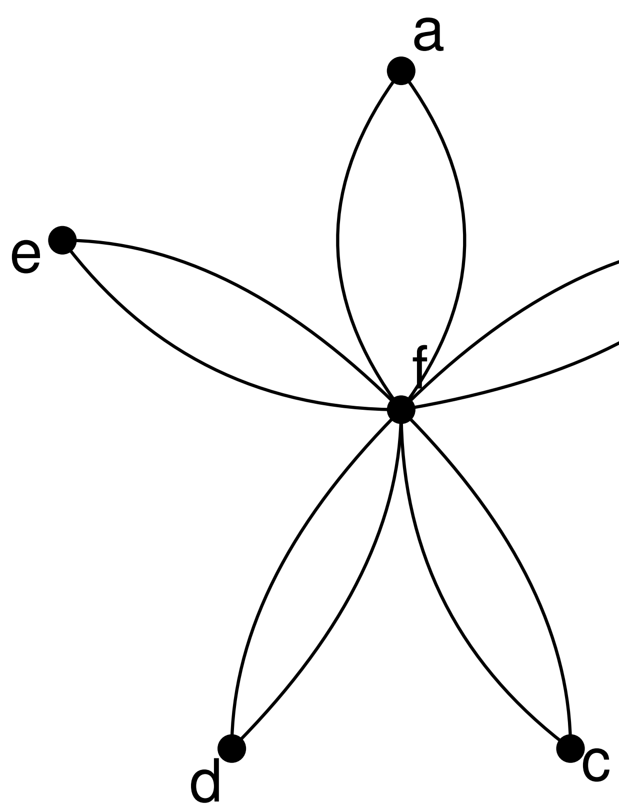
- given $G = (V, E, \omega)$, $\omega(e) = 1$
- metric completion, $\omega(e') = 2$
- compute MST T, $\omega(T) \leq \mathrm{OPT}$



Metric Traveling Salesman Problem

2-Approximation via MST

- given $G = (V, E, \omega)$, $\omega(e) = 1$
- metric completion, $\omega(e') = 2$
- compute MST T, $\omega(T) \leq \text{OPT}$
- double edges of T, $\omega(T') \leq 2\text{OPT}$

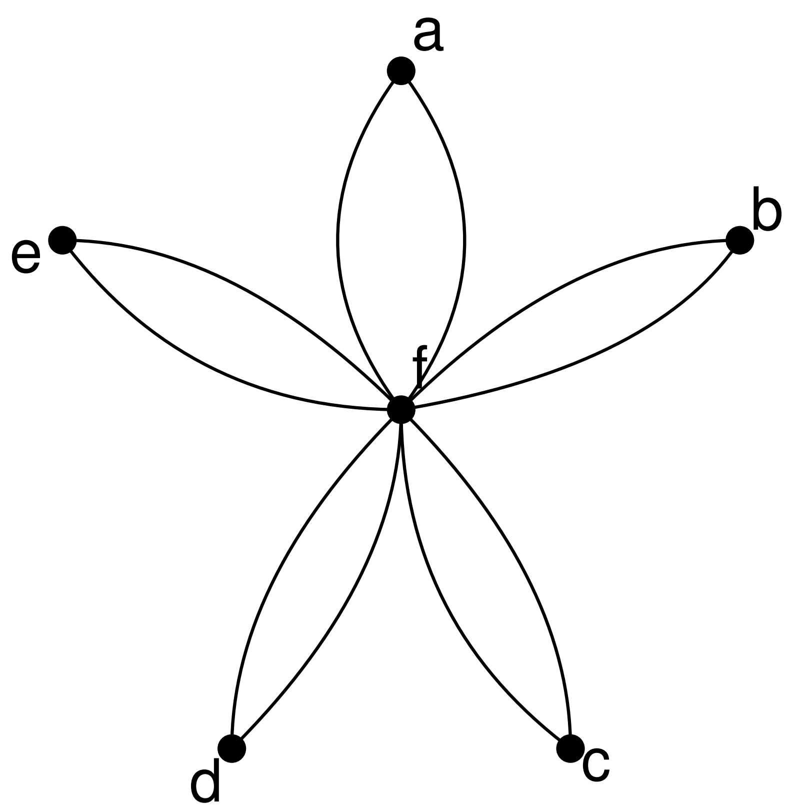


Metric Traveling Salesman Problem

2-Approximation via MST

- given $G = (V, E, \omega)$, $\omega(e) = 1$
- metric completion, $\omega(e') = 2$
- compute MST T, $\omega(T) \leq \text{OPT}$
- double edges of T, $\omega(T') \leq 2\text{OPT}$
- compute Eulerian tour

$$t = \{f, a, f, d, f, b, f, e, f, c, f\}$$



Metric Traveling Salesman Problem

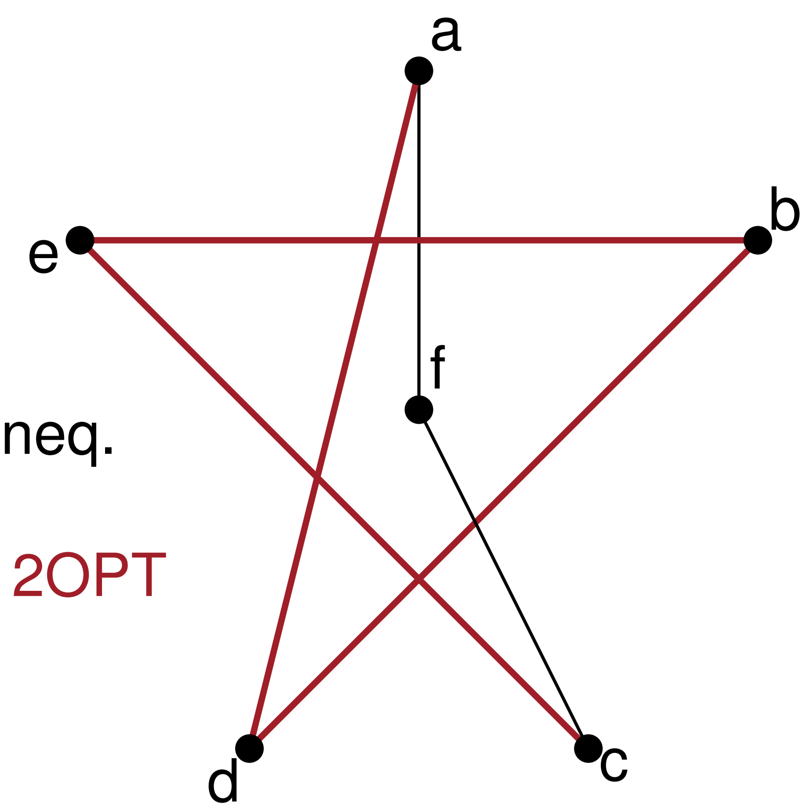
2-Approximation via MST

- given $G = (V, E, \omega)$, $\omega(e) = 1$
- metric completion, $\omega(e') = 2$
- compute MST T, $\omega(T) \leq \mathrm{OPT}$
- double edges of T, $\omega(T') \leq 2\mathrm{OPT}$
- compute Eulerian tour

$$t = \{f, a, f, d, f, b, f, e, f, c, f\}$$

- shortcut duplicates, $\omega(t') \leq \omega(t)$ tria. ineq.

$$t' = \{f, a, d, b, e, c, f\} \Rightarrow \omega(t') = 10 \leq 2\mathrm{OPT}$$



Metric Traveling Salesman Problem

Karlsruhe Institute of Technology

2-Approximation via MST

- given $G = (V, E, \omega)$, $\omega(e) = 1$
- metric completion, $\omega(e') = 2$
- compute MST T, $\omega(T) \leq \mathrm{OPT}$
- double edges of T, $\omega(T') \leq 2\mathrm{OPT}$
- compute Eulerian tour

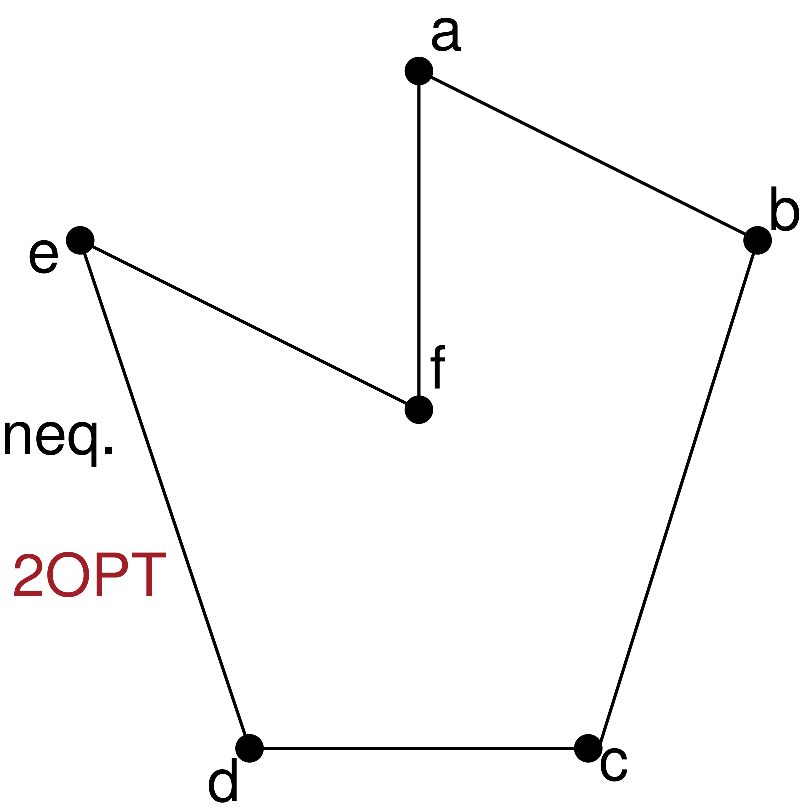
$$t = \{f, a, f, d, f, b, f, e, f, c, f\}$$

- shortcut duplicates, $\omega(t') \leq \omega(t)$ tria. ineq.

$$t' = \{f, a, d, b, e, c, f\} \Rightarrow \omega(t') = 10 \leq 2\mathrm{OPT}$$

- optimal tour

$$t^* = \{f, a, b, c, d, e, f\} \Rightarrow \omega(t^*) = 6$$



Metric Traveling Salesman Problem

2-Approximation via MST

- given $G = (V, E, \omega)$, $\omega(e) = 1$
- metric completion, $\omega(e') = 2$
- compute MST T, $\omega(T) \leq \mathrm{OPT}$
- double edges of T, $\omega(T') \leq 2\mathrm{OPT}$
- compute Eulerian tour

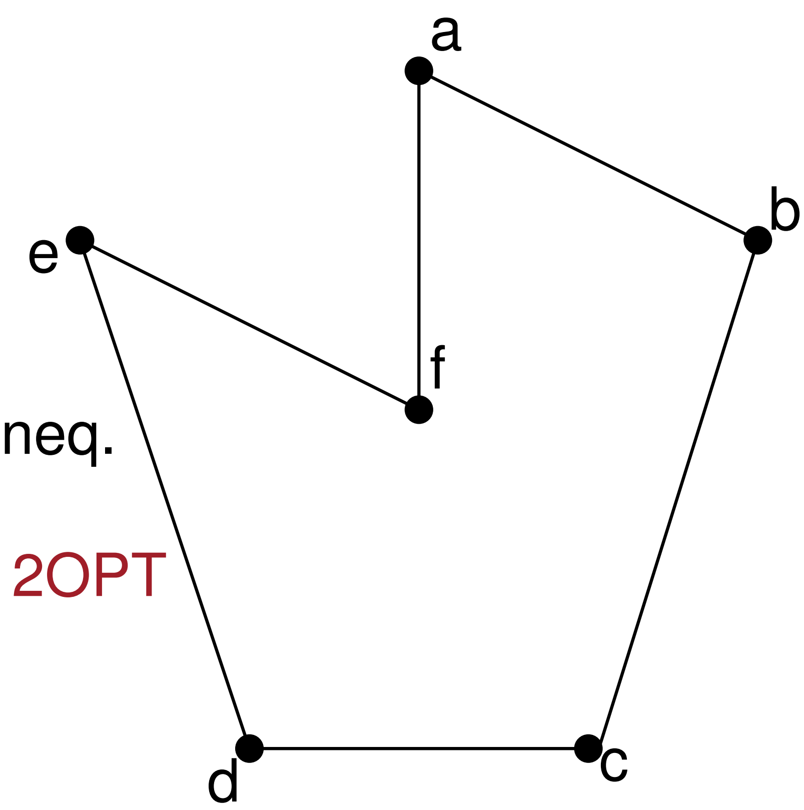
$$t = \{f, a, f, d, f, b, f, e, f, c, f\}$$

- shortcut duplicates, $\omega(t') \leq \omega(t)$ tria. ineq.

$$t' = \{f, a, d, b, e, c, f\} \Rightarrow \omega(t') = 10 \leq 2\mathrm{OPT}$$

- optimal tour

$$t^* = \{f, a, b, c, d, e, f\} \Rightarrow \omega(t^*) = 6$$



Good: $O\big(|E| + |V| \log |V|\big)$

Traveling Salesman Problem

- Metric TSP: $\frac{3}{2}$-approximation known
- Euclidean TSP: metric is Euclidean distance
  - Polynomial-time Approximation scheme (PTAS) known

Traveling Salesman Problem

Applications

- manifold applications in planning, logistics and manufacturing
- astronomy: minimize telescope movement between observed objects
- biology: matching genome sequences
- Vehicle Routing Problem: solve TSP for a fleet of vehicles
- Traveling Purchaser Problem: given different marketplaces
  find mimimum combined cost of traveling and purchasing a list of goods
- many more

Independent Sets

Given undirected Graph $G = (V, E)$

$I \subseteq V$ independent set $\Leftrightarrow$ no two vertices in I are adjacent in G

$I \subseteq V$ maximal independent set

$\Leftrightarrow$ I is no subset of any other independent set

Independent Sets

Given undirected Graph $G = (V, E)$

$I \subseteq V$ independent set $\Leftrightarrow$ no two vertices in I are adjacent in G

$I \subseteq V$ maximal independent set

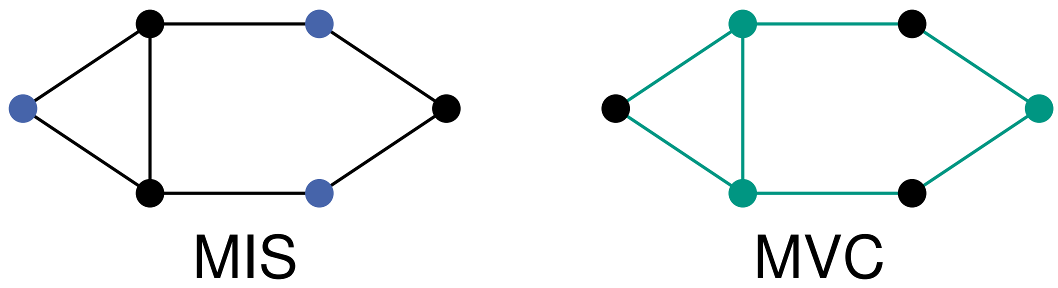
$\Leftrightarrow$ I is no subset of any other independent set

$I \subseteq V$ maximum independent set (MIS)

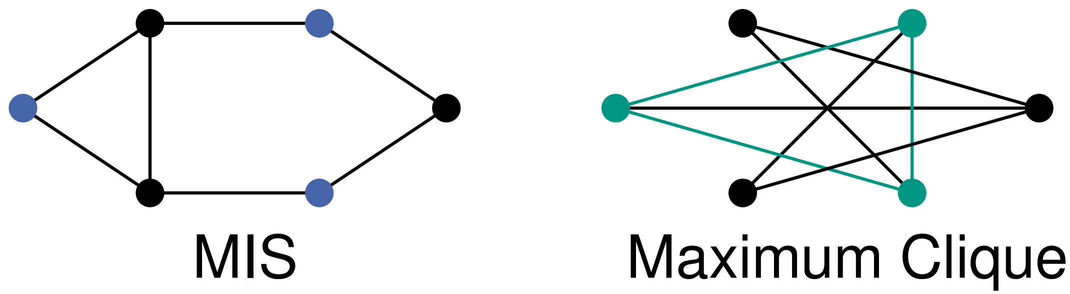
$\Leftrightarrow$ I is independent set with largest cardinality

Related Problems

- Vertex cover (VC): Find set of vertices that cover all edges
  ⇒ Complement of MIS is minimum vertex cover (MVC)



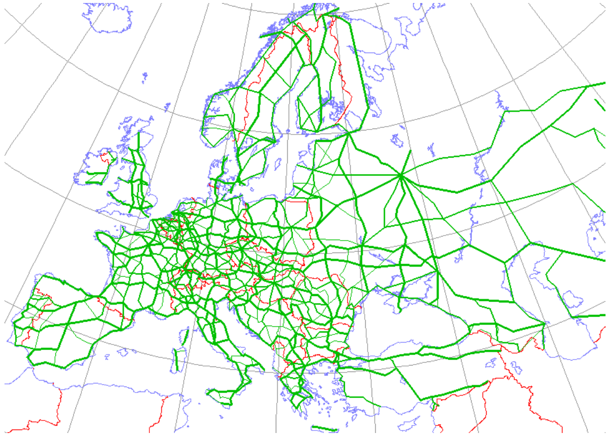
- Clique: Find set of vertices that are pairwise adjacent
  ⇒ MIS in complement graph is maximum clique



Institute of Theoretical Informatics
Algorithmics Group

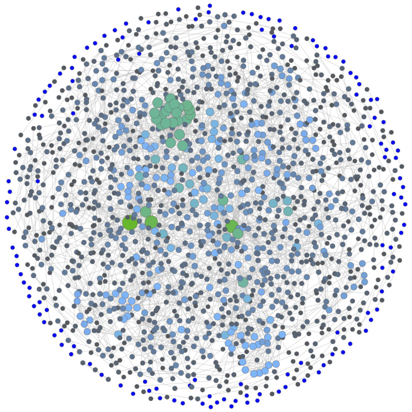
Applications

Partitioning of social networks



Map labeling/shortest-path computations

Mesh edge ordering in rendering



Finding protein-protein interactions

Finding Maximum Independent Sets

- Find independent set with maximum number of vertices (MIS)
  $\Rightarrow$ Optimization problem is NP-hard

Finding Maximum Independent Sets

- Find independent set with maximum number of vertices (MIS)
  $\Rightarrow$ Optimization problem is NP-hard

- Exact algorithms in general graphs
  - Brute-force algorithm in $\mathcal{O}(n^2 2^n)$
  - Best exact algorithm with polynomial space in $\mathcal{O}(1.1996^n)$

Finding Maximum Independent Sets

- Find independent set with maximum number of vertices (MIS)
  $\Rightarrow$ Optimization problem is NP-hard

- Exact algorithms in general graphs
  - Brute-force algorithm in $\mathcal{O}(n^2 2^n)$
  - Best exact algorithm with polynomial space in $\mathcal{O}(1.1996^n)$

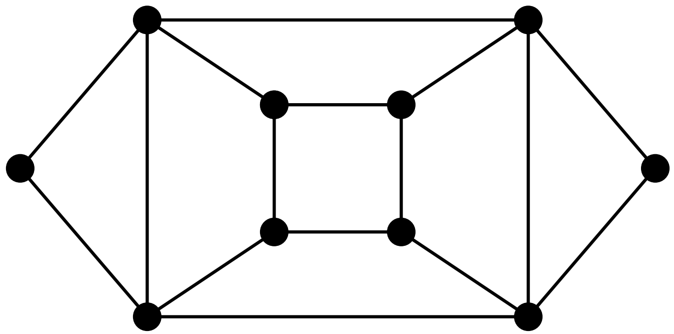
- Even worse for general graphs
  - No constant factor approximations in polynomial time
  - Polynomial time approximations for planar and unit disk graphs

$\Rightarrow$ Ugly How to find good heuristics?

Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

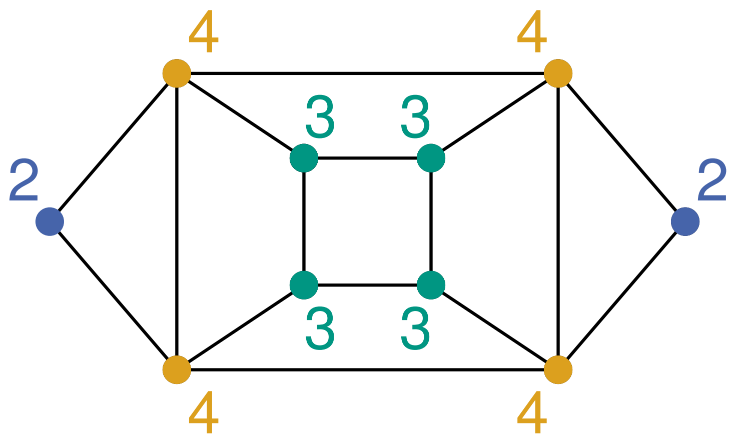
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

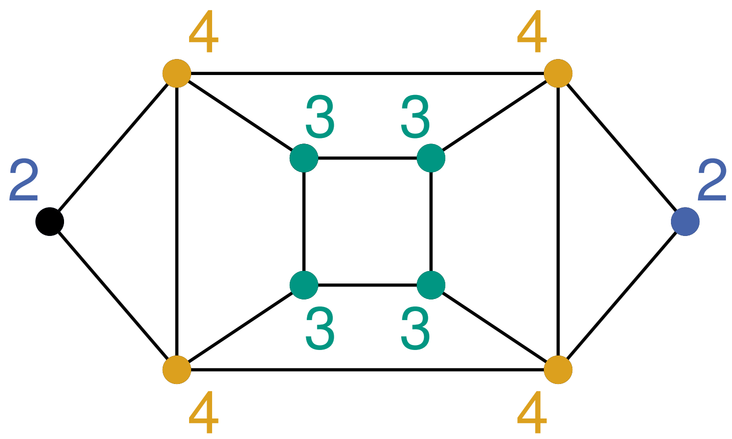
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

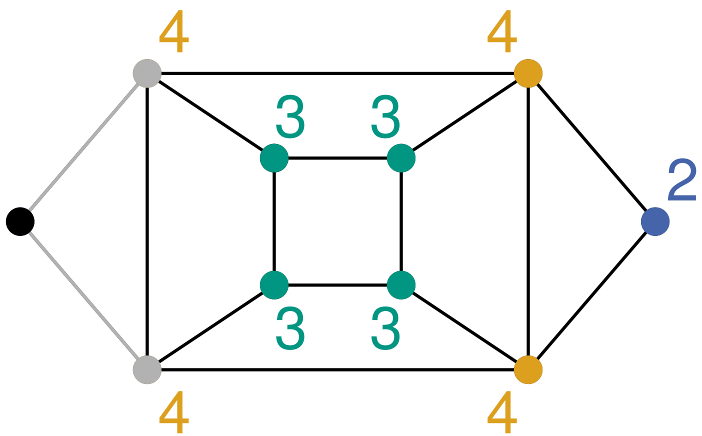
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

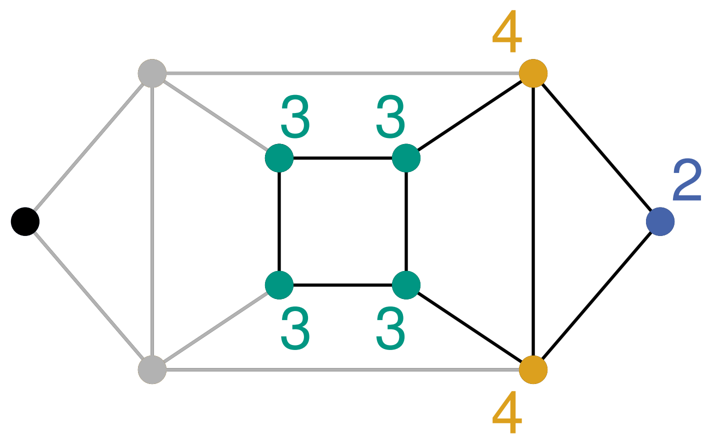
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

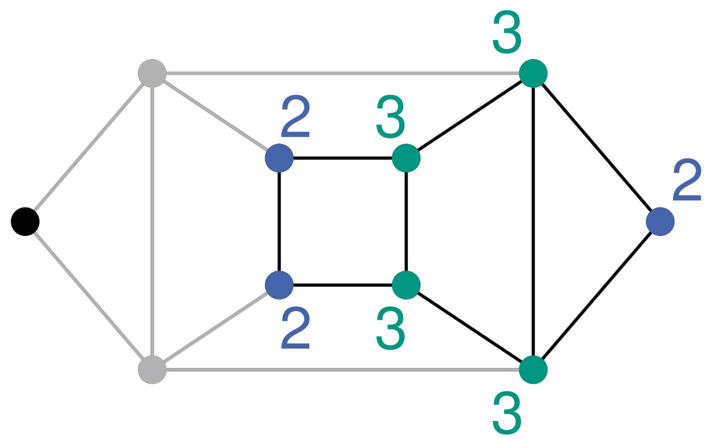
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

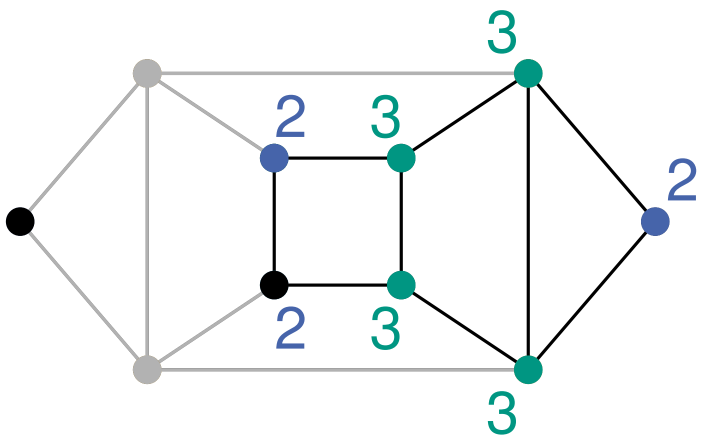
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

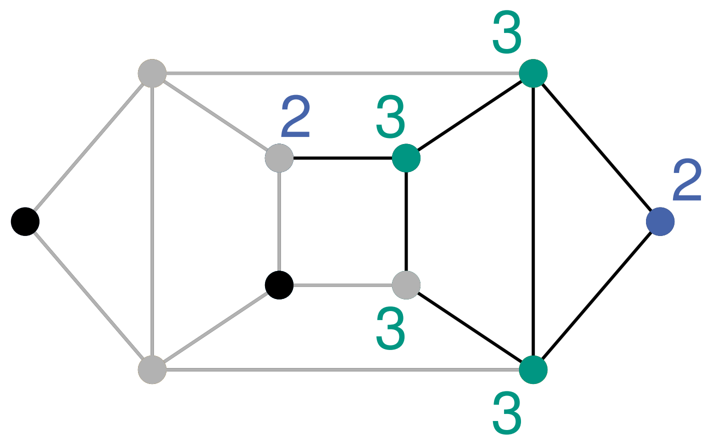
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

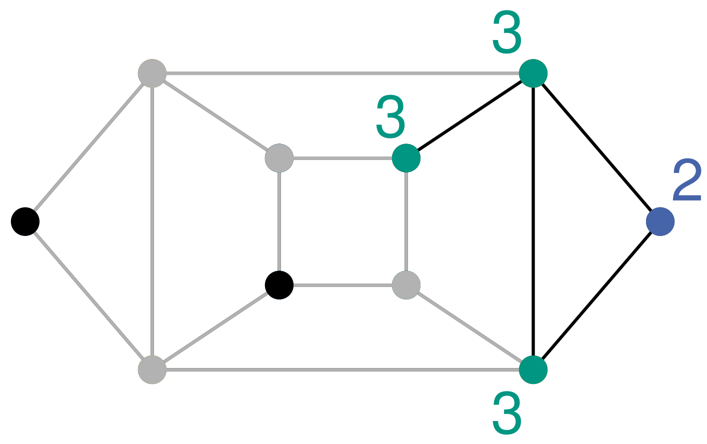
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors

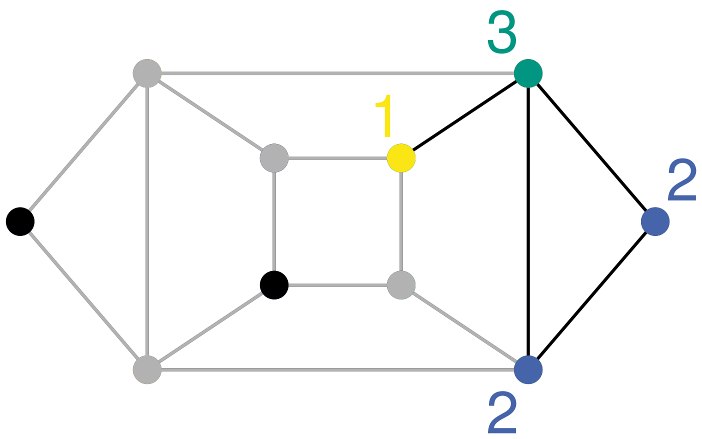


Greedy Heuristic

Karlsruhe Institute of Technology

Given undirected Graph $G = (V, E)$ with bounded degree Δ

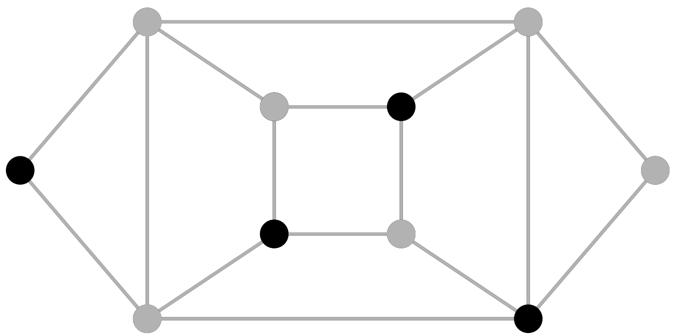
1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors



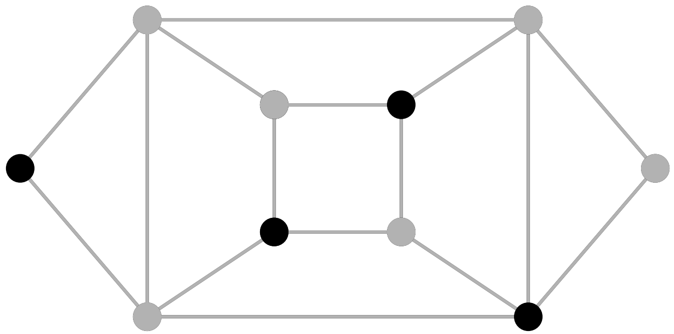
Greedy Heuristic

Given undirected Graph $G = (V, E)$ with bounded degree Δ

1. Sort vertices in buckets by ascending degree
2. Vertices remaining?
   (a) Select random vertex from bucket with lowest degree
   (b) Add vertex to independent set
   (c) Remove neighboring vertices
   (d) Decrease degree of next neighbors

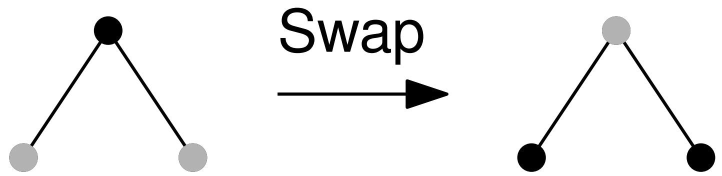
$\Rightarrow \frac{\Delta+2}{3}$ approximation

Good $\mathcal{O}(n+m)$



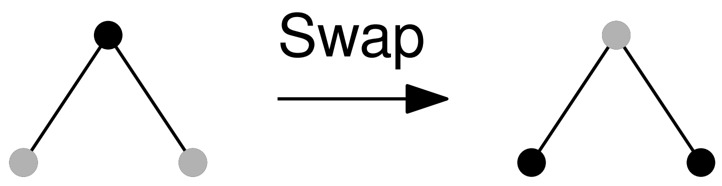
Finding Independent Sets in Practice

- Local Search
  - Swap vertices to gradually find better solutions
  - Use different diversification methods

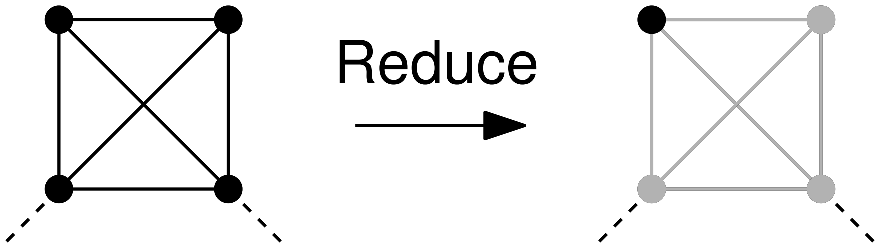


Finding Independent Sets in Practice

- Local Search
  - Swap vertices to gradually find better solutions
  - Use different diversification methods



- Reductions
  - Find vertices that are contained in any maximum independent set
  - Remove vertices to reduce problem size



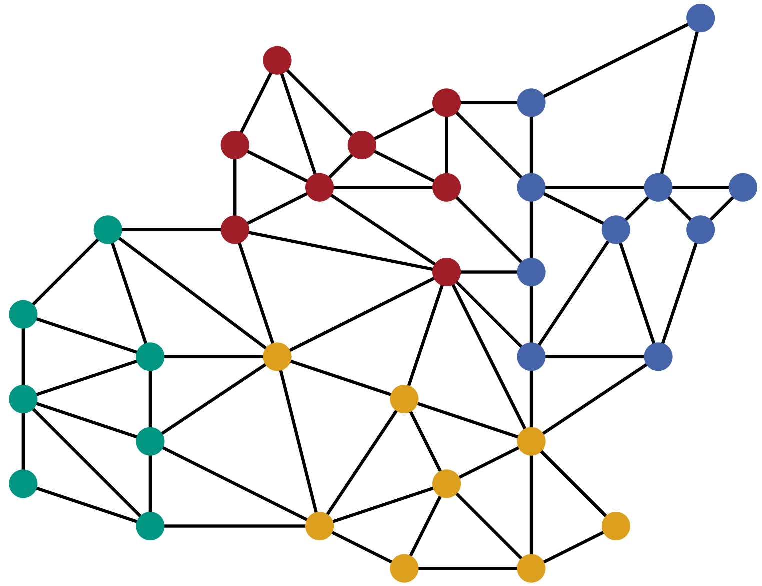
ε-Balanced Graph and Hypergraph Partitioning

Partition (hyper)graph $G = (V, E, c : V \to \mathbb{R}_{>0}, \omega : E \to \mathbb{R}_{>0})$ into **k** disjoint blocks $V_1, \ldots, V_k$ s.t.

- blocks V_i are **roughly equal-sized**:

$$c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$$

- **objective** function on edges is **minimized**



ε-Balanced Graph and Hypergraph Partitioning

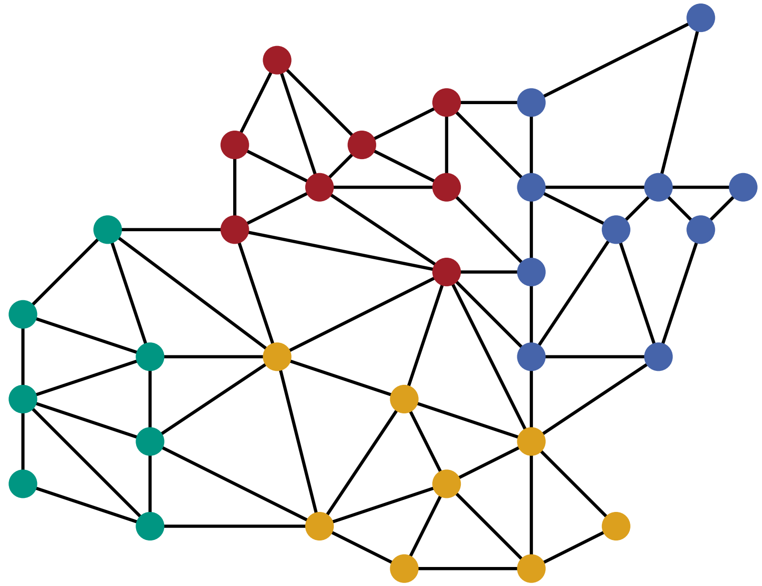
Partition (hyper)graph $G = (V, E, c : V \to \mathbb{R}_{>0}, \omega : E \to \mathbb{R}_{>0})$ into **k** disjoint blocks $V_1, \ldots, V_k$ s.t.

- blocks V_i are **roughly equal-sized**:

imbalance parameter

$$c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$$

- **objective** function on edges is **minimized**



Institute of Theoretical Informatics
Algorithmics Group

ε-Balanced Graph and Hypergraph Partitioning

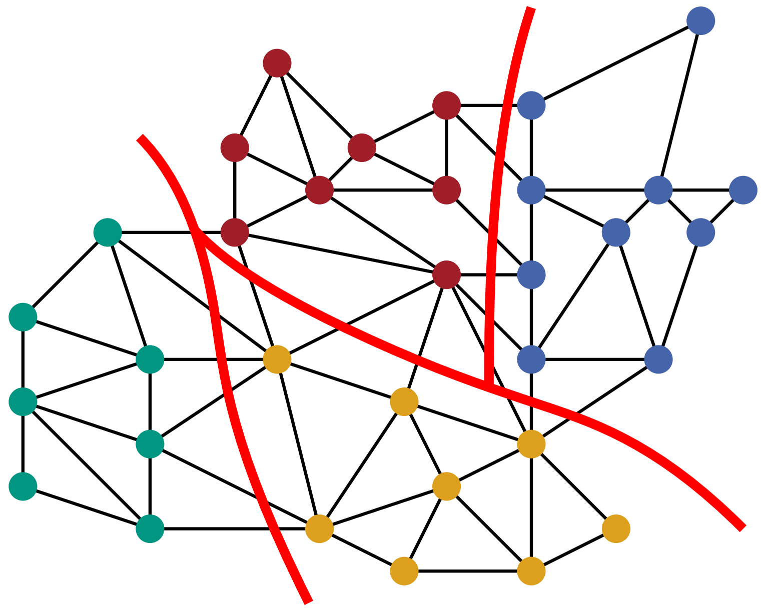
Partition (hyper)graph $G = (V, E, c : V \to \mathbb{R}_{>0}, \omega : E \to \mathbb{R}_{>0})$ into **k** disjoint blocks $V_1, \ldots, V_k$ s.t.

- blocks V_i are **roughly equal-sized**:

imbalance parameter

$$c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$$

- **objective** function on edges is **minimized**



Institute of Theoretical Informatics
Algorithmics Group

ε-Balanced Graph and Hypergraph Partitioning

Partition (hyper)graph $G = (V, E, c : V \to \mathbb{R}_{>0}, \omega : E \to \mathbb{R}_{>0})$ into **k** disjoint blocks $V_1, \ldots, V_k$ s.t.

- blocks V_i are **roughly equal-sized**:

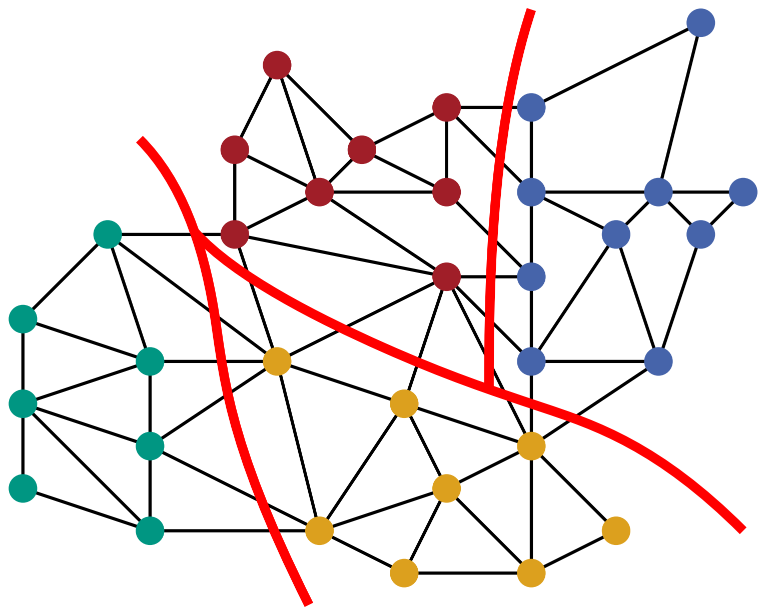
imbalance parameter

$$c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$$

- **objective** function on edges is **minimized**

Common Objectives:

- Graphs:
  - cut: $\sum_{e \in \text{cut}} \omega(e)$



ε-Balanced Graph and Hypergraph Partitioning

Partition (hyper)graph $G = (V, E, c : V \to \mathbb{R}_{>0}, \omega : E \to \mathbb{R}_{>0})$ into **k** disjoint blocks $V_1, \ldots, V_k$ s.t.

- blocks V_i are **roughly equal-sized**:

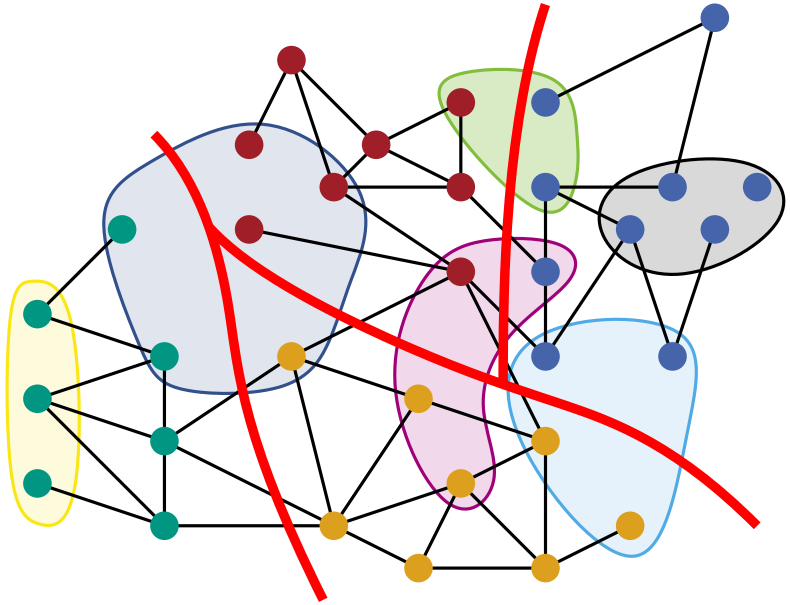
imbalance parameter

$$c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$$

- **objective** function on edges is **minimized**

Common Objectives:

- Graphs:
  - cut: $\sum_{e \in \text{cut}} \omega(e)$



ε-Balanced Graph and Hypergraph Partitioning

Partition (hyper)graph $G = (V, E, c : V \to \mathbb{R}_{>0}, \omega : E \to \mathbb{R}_{>0})$ into **k** disjoint blocks $V_1, \ldots, V_k$ s.t.

- blocks V_i are **roughly equal-sized**:

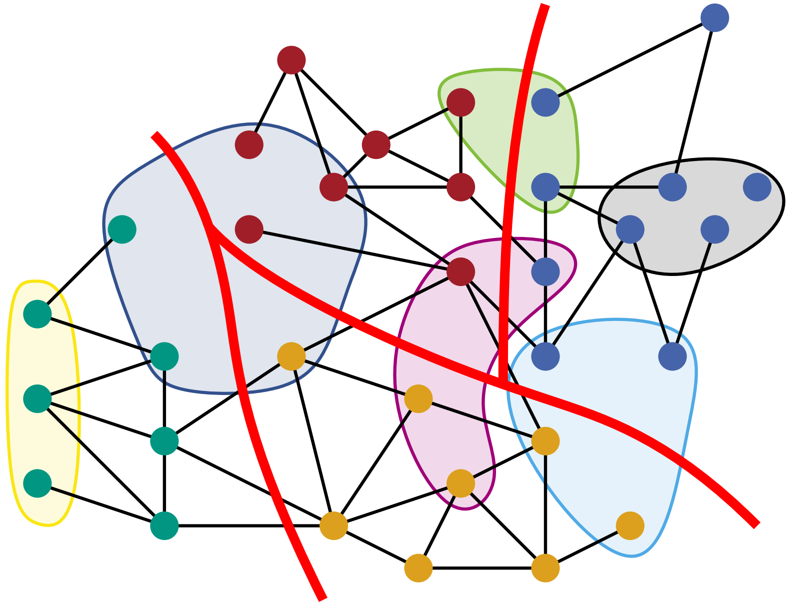
imbalance parameter

$$c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$$

- **objective** function on edges is **minimized**

Common Objectives:

- Graphs:
  - cut: $\sum_{e \in \text{cut}} \omega(e)$
- Hypergraphs:
  - cut: $\sum_{e \in \text{cut}} \omega(e)$
  - connectivity: $\sum_{e \in \text{cut}} (\lambda - 1)\, \omega(e)$



ε-Balanced Graph and Hypergraph Partitioning

Partition (hyper)graph $G = (V, E, c : V \to \mathbb{R}_{>0}, \omega : E \to \mathbb{R}_{>0})$ into **k** disjoint blocks $V_1, \ldots, V_k$ s.t.

- blocks V_i are **roughly equal-sized**:

imbalance parameter

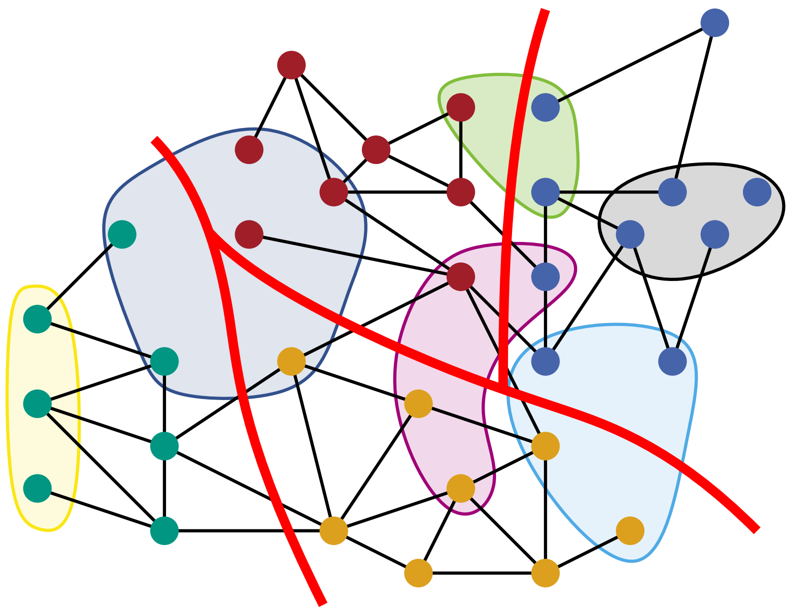
$$c(V_i) \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil$$

- **objective** function on edges is **minimized**

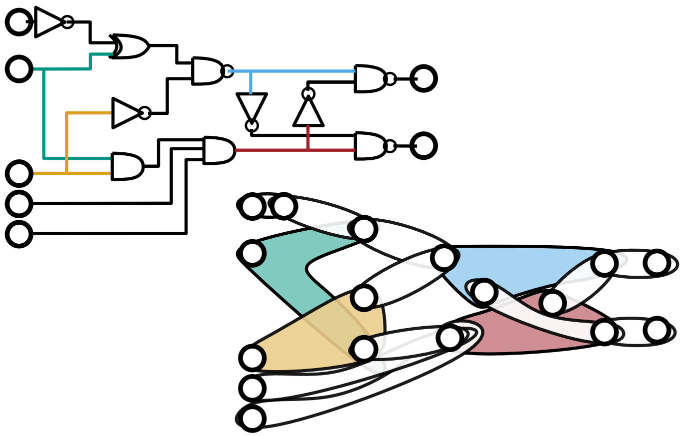
Common Objectives:

- Graphs:
  - cut: $\sum_{e \in \text{cut}} \omega(e)$
- Hypergraphs:
  - cut: $\sum_{e \in \text{cut}} \omega(e)$
  - connectivity: $\sum_{e \in \text{cut}} (\lambda - 1)\, \omega(e)$

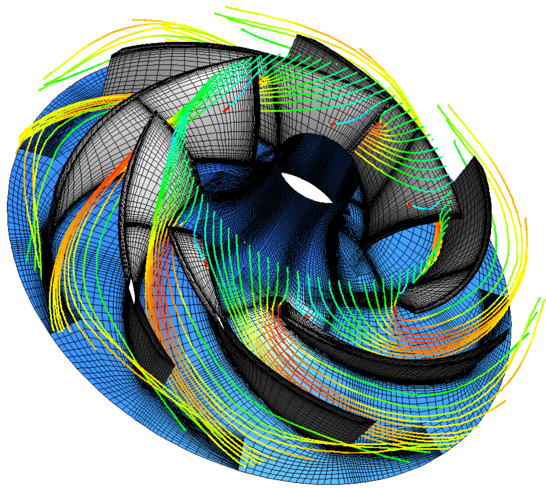
 # blocks connected by e

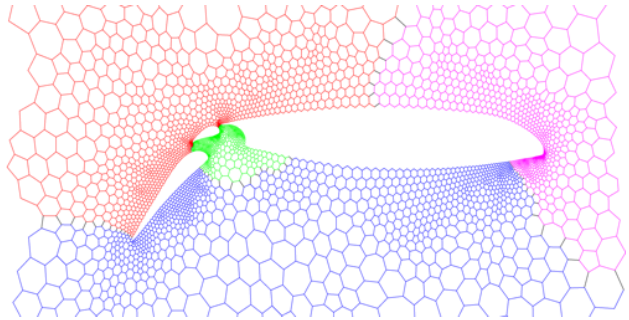


Applications



VLSI Design **Warehouse Optimization** **Complex Networks**





$$\mathbb{R}^{n\times n} \ni Ax = b \in \mathbb{R}^n$$

Route Planning **Simulation** **Scientific Computing**

(Hyper)Graph Partitioning Algorithms

- Hypergraph Partitioning is $\mathbf{NP}$-hard
- even finding good approximate solutions for graphs is $\mathbf{NP}$-hard

Institute of Theoretical Informatics
Algorithmics Group

(Hyper)Graph Partitioning Algorithms

- Hypergraph Partitioning is $\mathbf{NP}$-hard
- even finding good approximate solutions for graphs is $\mathbf{NP}$-hard

$\Rightarrow$ **exact** solutions only for very small graphs & small *k* feasible!
$\Rightarrow$ most **successful** heuristic: **Multilevel Approach**

(Hyper)Graph Partitioning Algorithms

- Hypergraph Partitioning is $\mathbf{NP}$-hard
- even finding good approximate solutions for graphs is $\mathbf{NP}$-hard

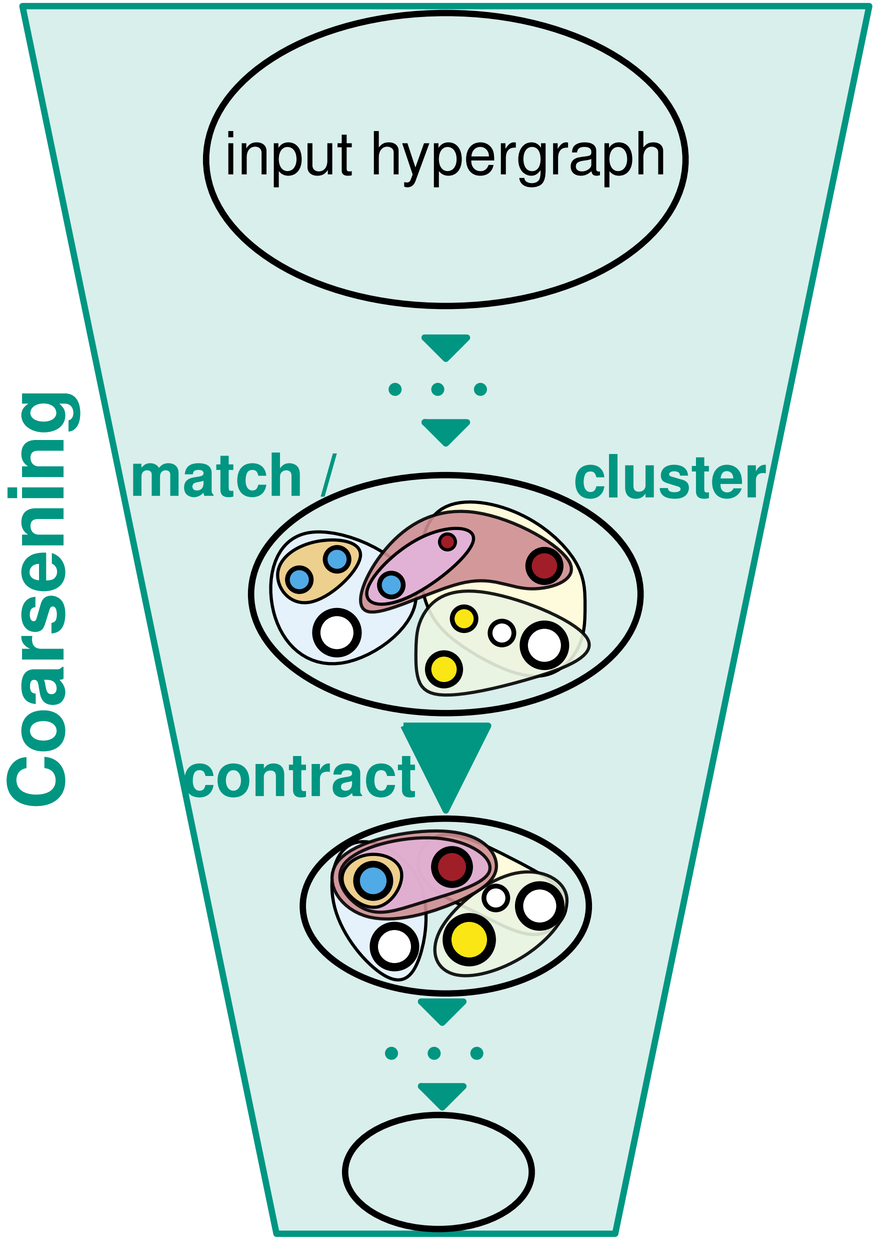
Ugly: $\mathbf{NP}$-hard, not $\mathbf{APX}$

$\Rightarrow$ **exact** solutions only for very small graphs & small *k* feasible!
$\Rightarrow$ most **successful** heuristic: **Multilevel Approach**

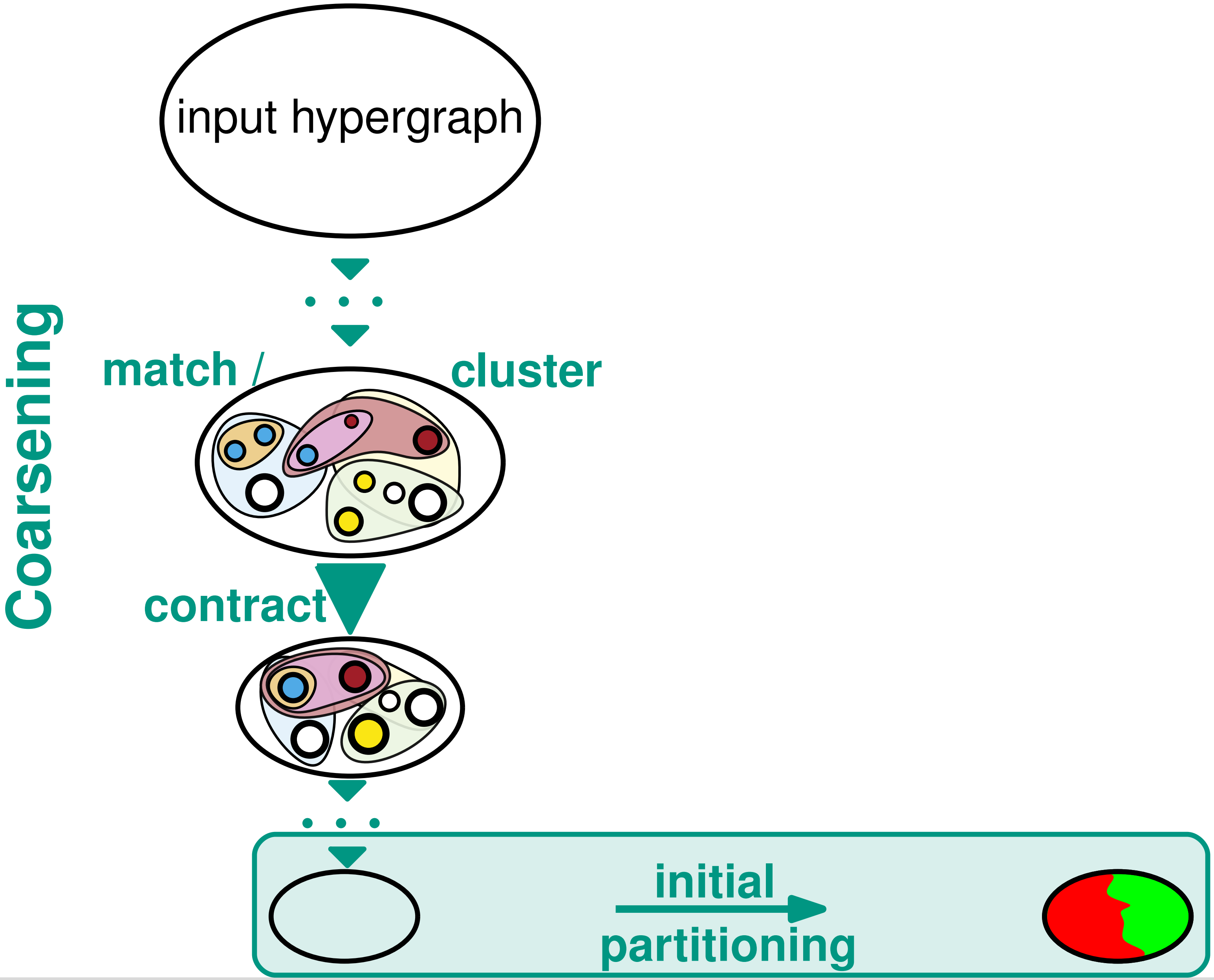
Sophisticated partitioners developed in our group:

- **KaHIP** – Karlsruhe High Quality Partitioning
  - Objective: cut
  - `https://git.io/vderw`
- **KaHyPar** – Karlsruhe Hypergraph Partitioning
  - Objectives: cut, $(\lambda - 1)$
  - `https://git.io/vMBaR`

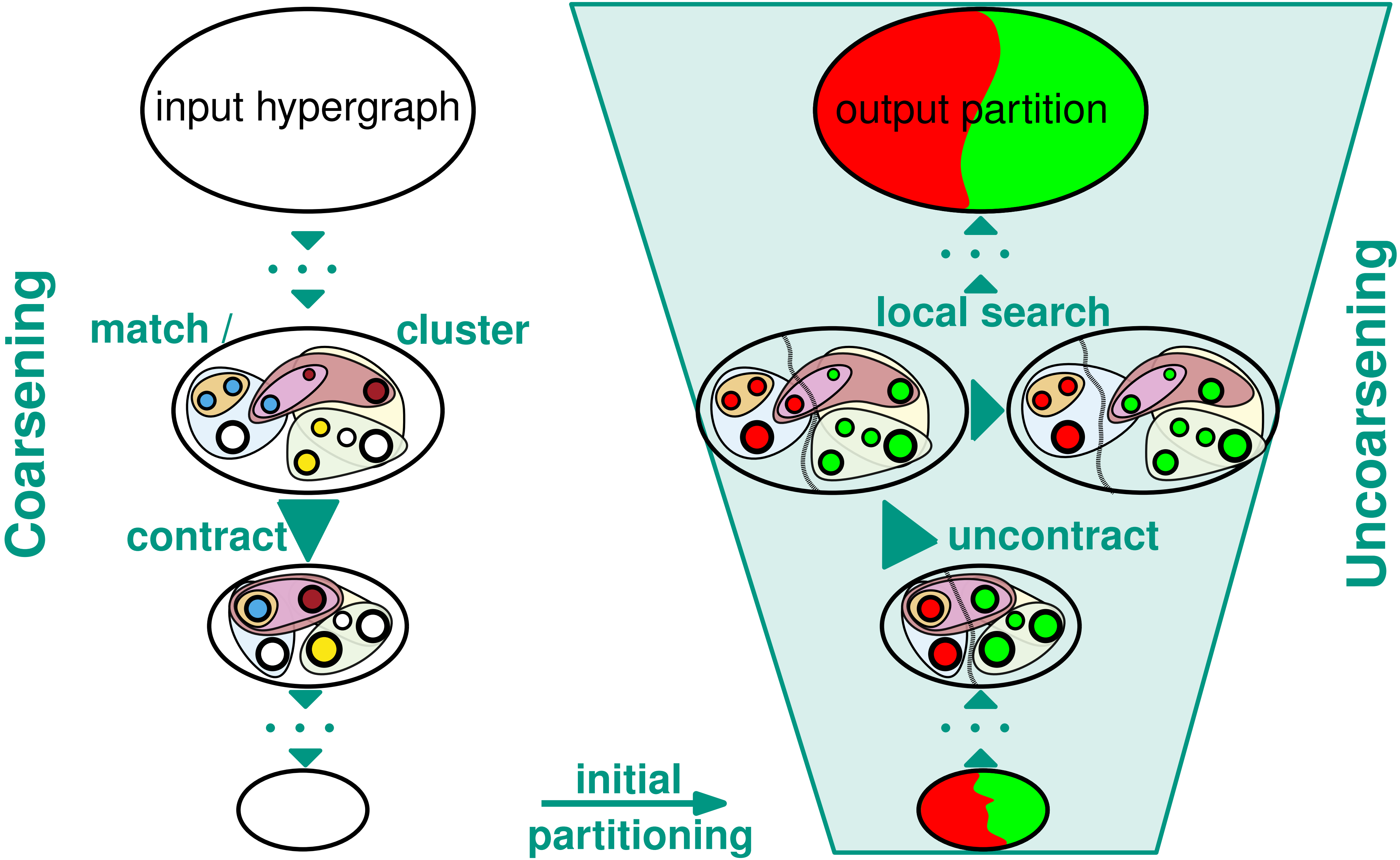
Multilevel Paradigm



Multilevel Paradigm



Multilevel Paradigm



Hill Climbing vs. Local Search

Hill Climbing

find some feasible solution $x \in \mathcal{L}$

$\bar{x} \leftarrow x$ ▷ best solution found so far

while true do

  if $\exists x \in \mathcal{N}(x) \cap \mathcal{L} : f(x) < f(\hat{x})$ **then**

    $\bar{x} \leftarrow x$

  else

    return $\bar{x}$ ▷ local optimum found

Local Search

find some feasible solution $x \in \mathcal{L}$

$\hat{x} \leftarrow x$ ▷ $\hat{x}$ is best solution found so far

while not satisfied with $\hat{x}$ **do**

  $x \leftarrow$ some heuristically chosen element from $\mathcal{N}(x) \cap \mathcal{L}$

  if $f(x) < f(\hat{x})$ **then**

    $\hat{x} \leftarrow x$

Hill Climbing vs. Local Search

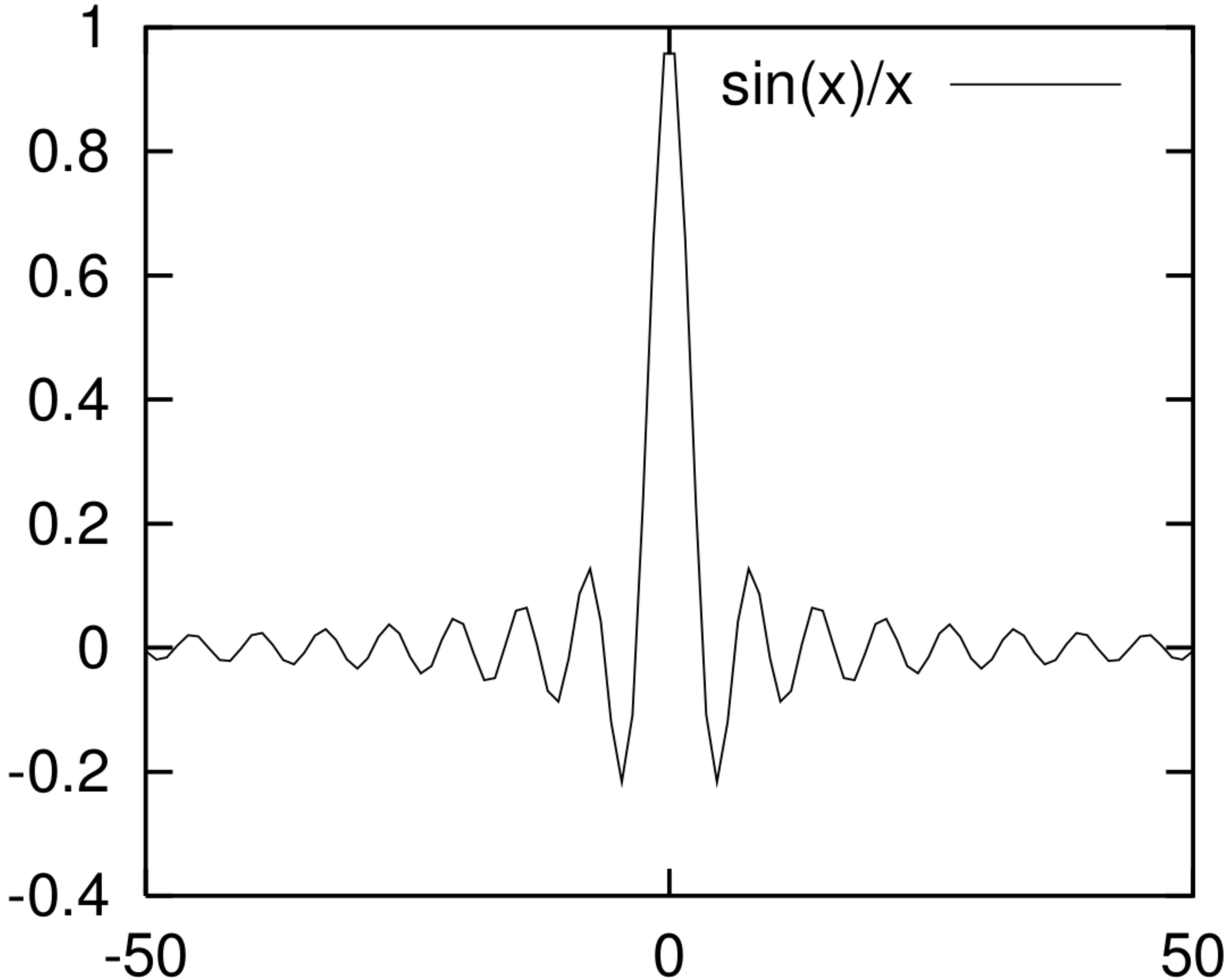
Hill
fir
$\bar{x}$
wl

nd so far

ım found

Loca
fir
$\hat{x}$
wl

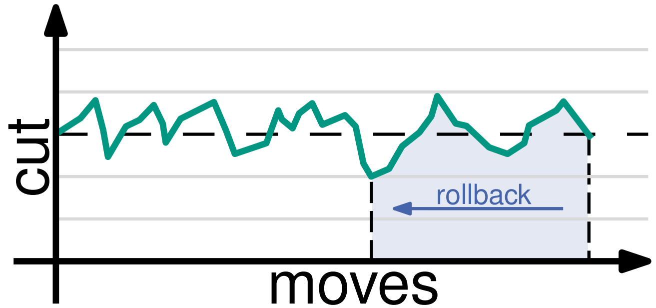
ınd so far



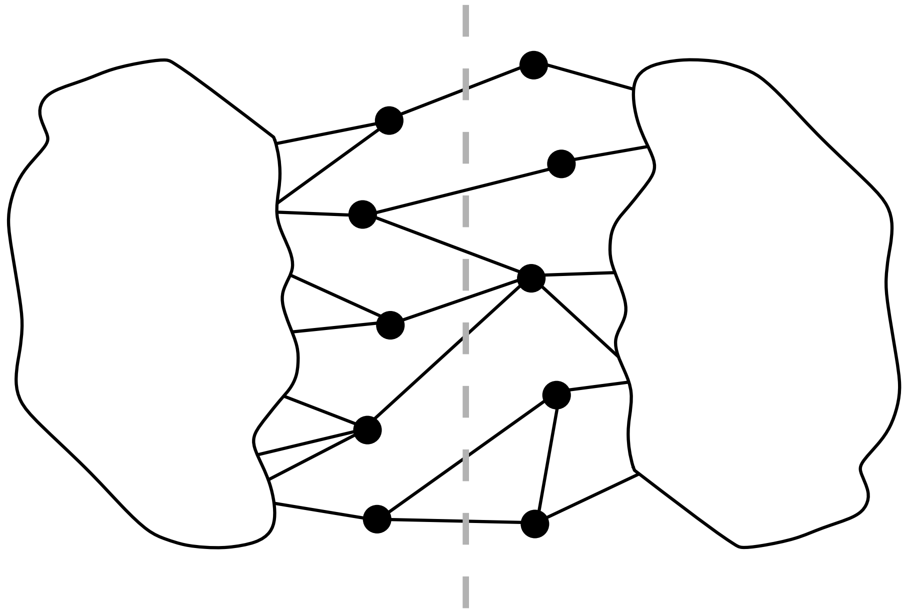
Fiduccia-Mattheyses Algorithm

FM Local Search

while ¬ done **do**
  find best move
  perform best move
rollback to best solution



can worsen solution



- compute gain $g(v) = d_{\text{ext}}(v) - d_{\text{int}}(v)$
- alternate between blocks
- edge-cut: **7**

Fiducca-Mattheyses Algorithm

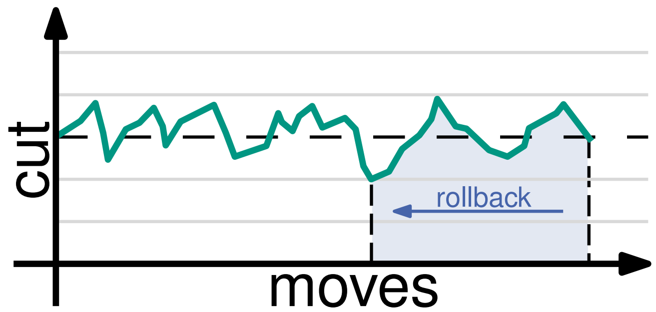
FM Local Search

while $\neg$ done **do**

find best move

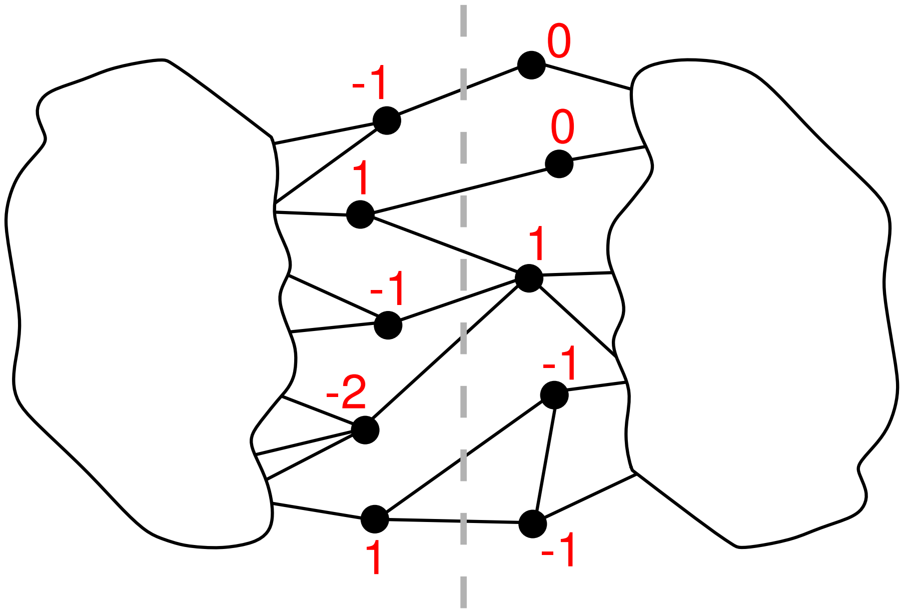
perform best move

rollback to best solution



can worsen solution

- compute gain $g(v) = d_{\text{ext}}(v) - d_{\text{int}}(v)$
- alternate between blocks
- edge-cut: **7**

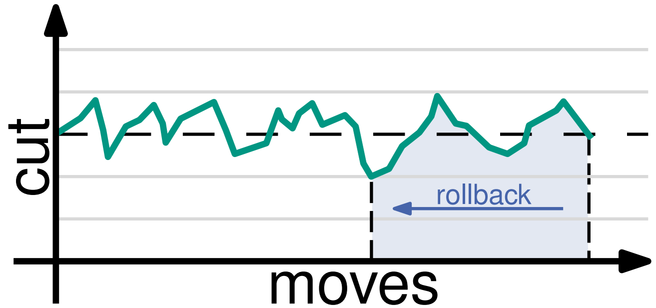


Fiduccia-Mattheyses Algorithm

Karlsruhe Institute of Technology

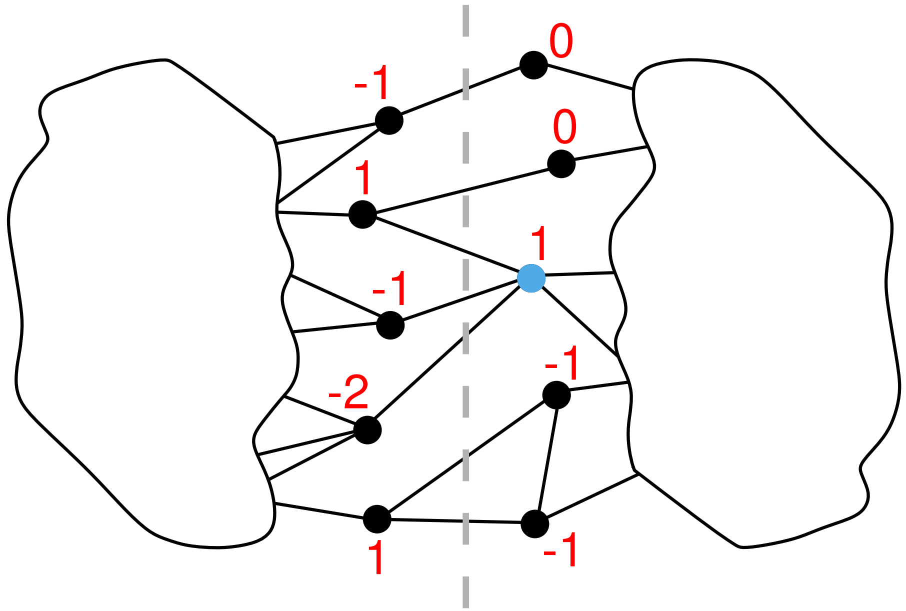
FM Local Search

while ¬ done **do**
find best move
perform best move
rollback to best solution



can worsen solution

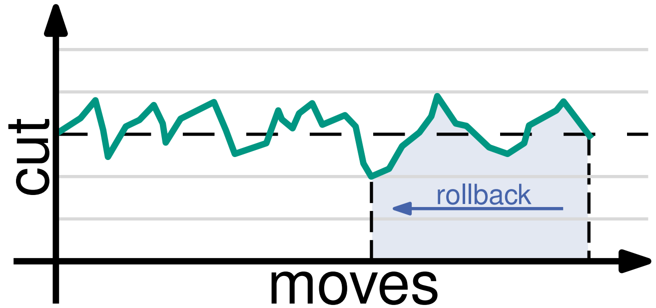
- compute gain $g(v) = d_{ext}(v) - d_{int}(v)$
- alternate between blocks
- edge-cut: **7**



Fiduccia-Mattheyses Algorithm

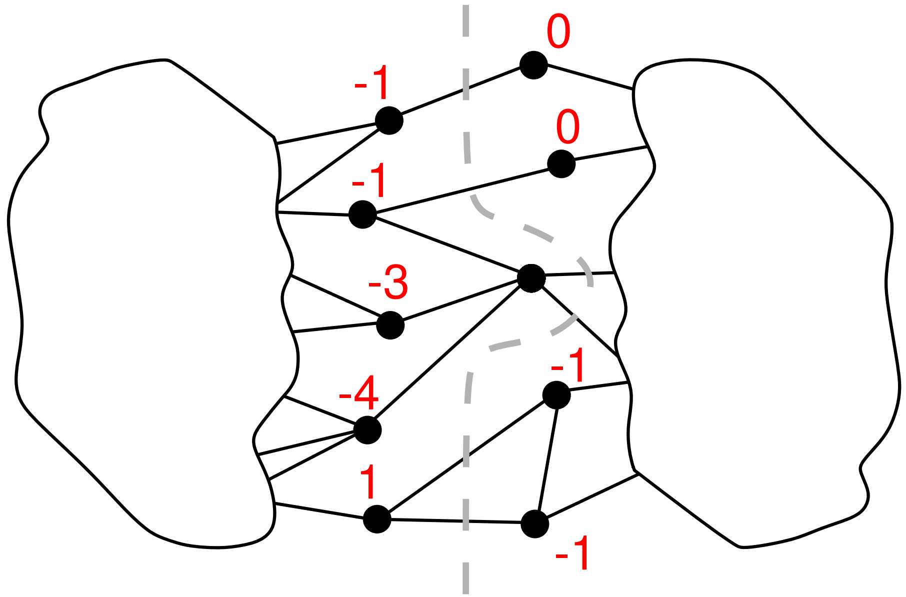
FM Local Search

while ¬ done **do**
  find best move
  perform best move
rollback to best solution



can worsen solution

- recalculate gain $g(v)$ of neighbors
- move each node at most once
- edge-cut: **7, 6**

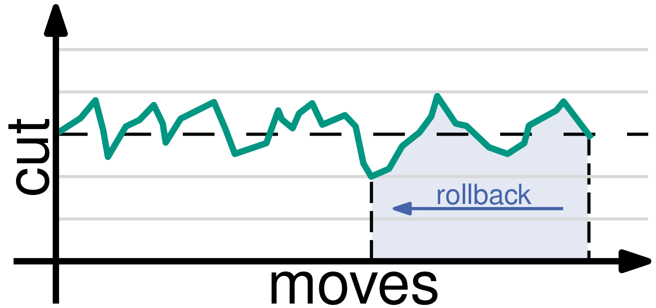


Institute of Theoretical Informatics
Algorithmics Group

Fiduccia-Mattheyses Algorithm

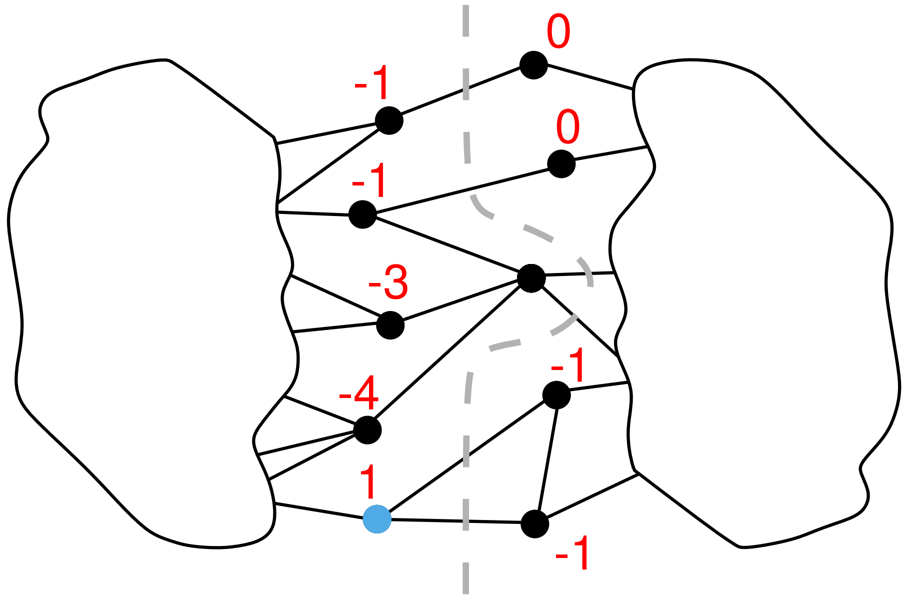
FM Local Search

while ¬ done **do**
find best move
perform best move
rollback to best solution



can worsen solution

- recalculate gain $g(v)$ of neighbors
- move each node at most once
- edge-cut: **7, 6**

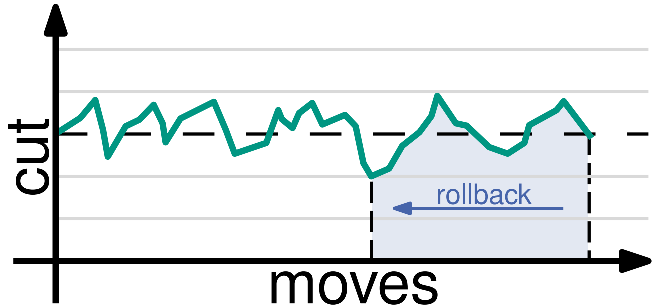


Institute of Theoretical Informatics
Algorithmics Group

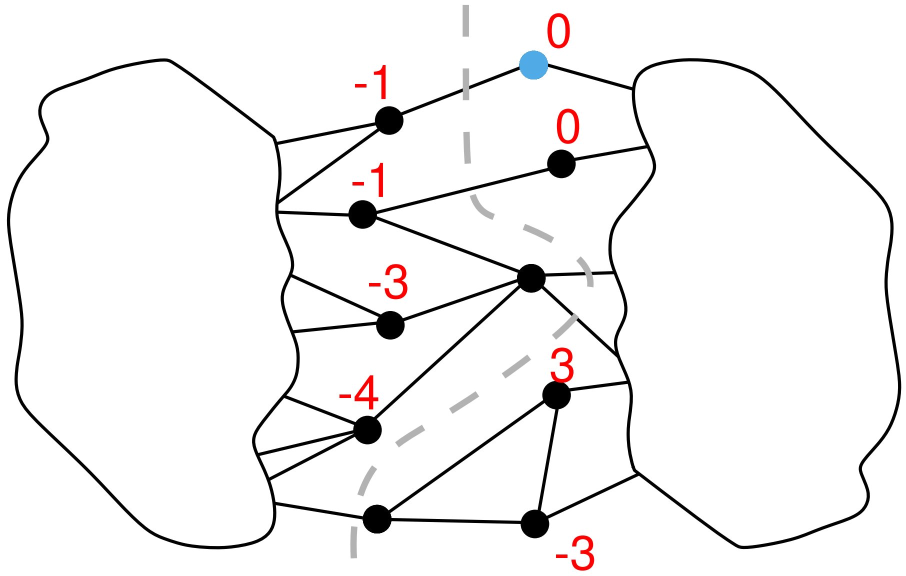
Fiduccia-Mattheyses Algorithm

FM Local Search

while ¬ done **do**
  find best move
  perform best move
rollback to best solution



can worsen solution



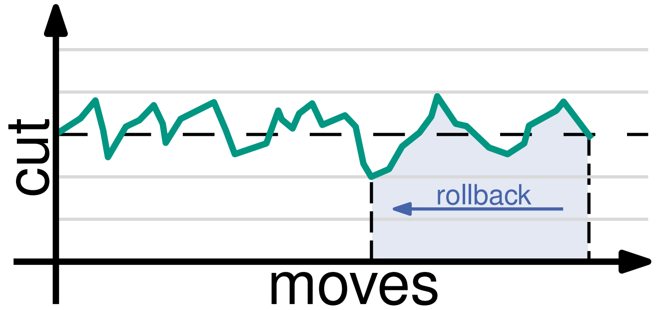
- recalculate gain $g(v)$ of neighbors
- move each node at most once
- edge-cut: **7, 6,5**

Institute of Theoretical Informatics
Algorithmics Group

Fiduccia-Mattheyses Algorithm

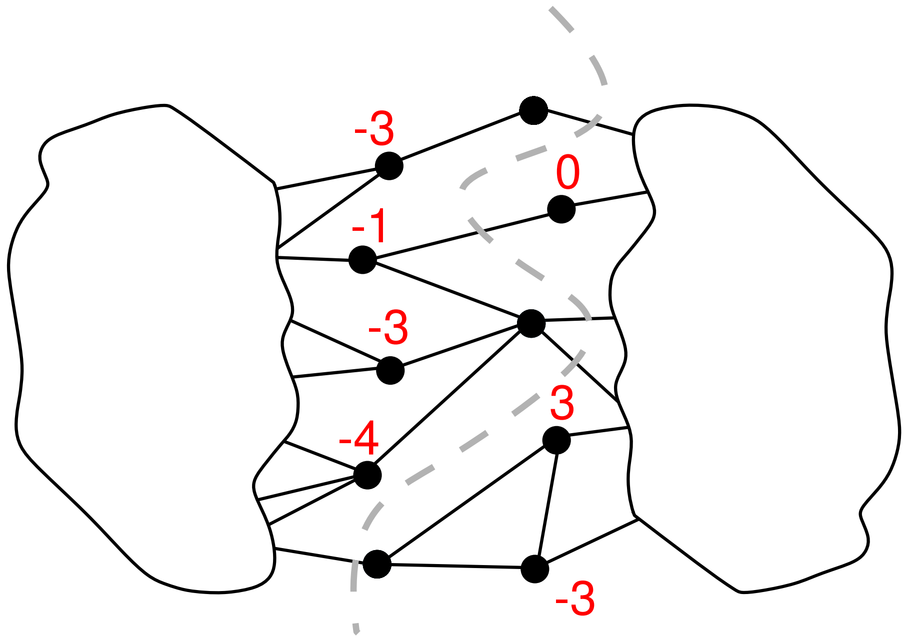
FM Local Search

while ¬ done **do**
find best move
perform best move
rollback to best solution



can worsen solution

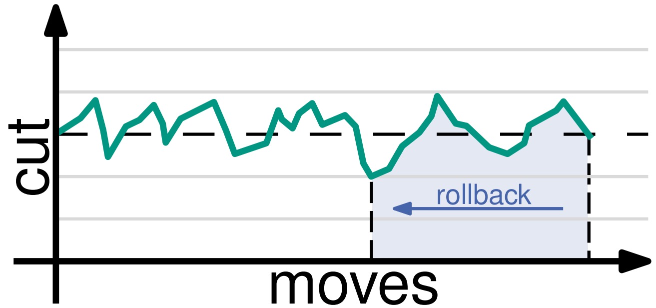
- recalculate gain $g(v)$ of neighbors
- move each node at most once
- edge-cut: **7, 6,5,5**



Fiduccia-Mattheyses Algorithm

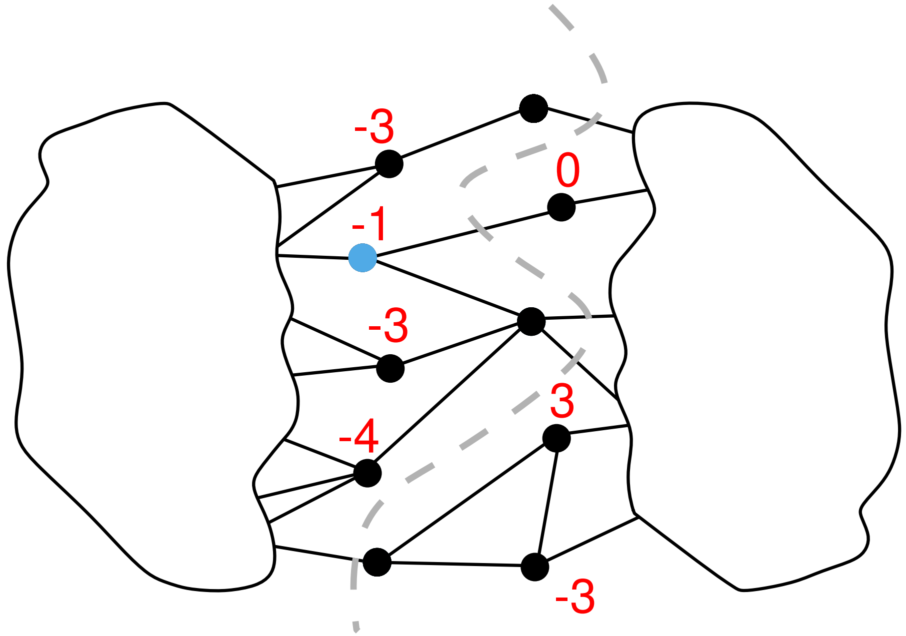
FM Local Search

while ¬ done **do**
    find best move
    perform best move
rollback to best solution



can worsen solution

- recalculate gain $g(v)$ of neighbors
- move each node at most once
- edge-cut: **7, 6,5,5**

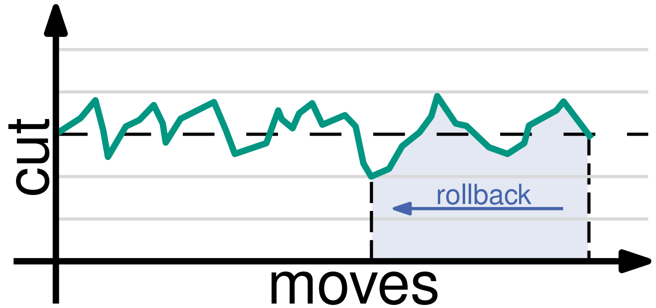


Institute of Theoretical Informatics
Algorithmics Group

Fiduccia-Mattheyses Algorithm

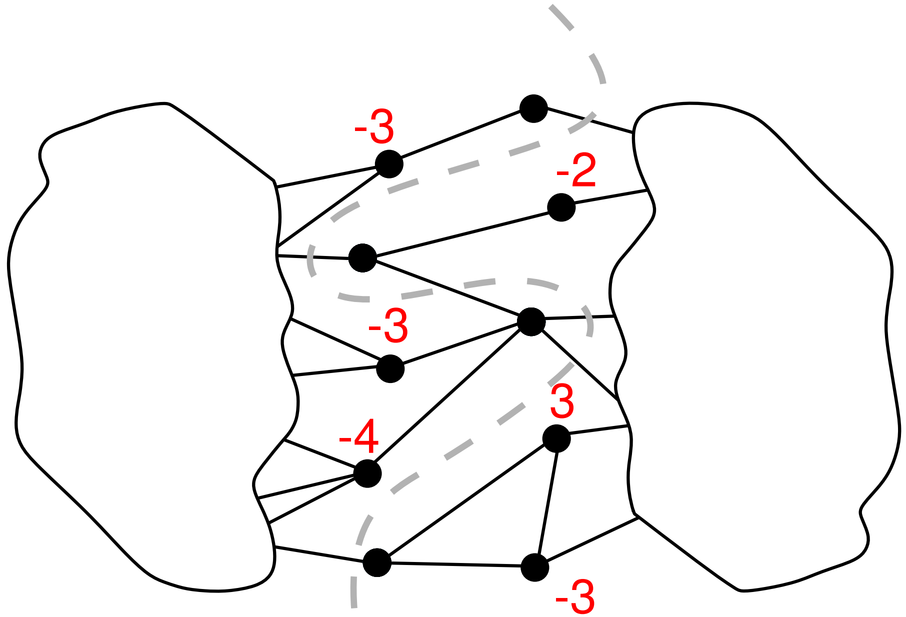
FM Local Search

while ¬ done **do**
  find best move
  perform best move
rollback to best solution



can worsen solution

- recalculate gain $g(v)$ of neighbors
- move each node at most once
- edge-cut: **7, 6,5,5,6**

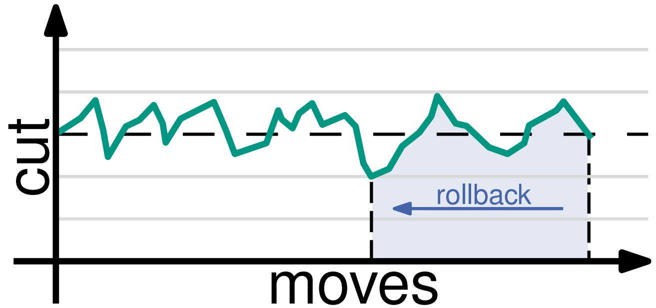


Institute of Theoretical Informatics
Algorithmics Group

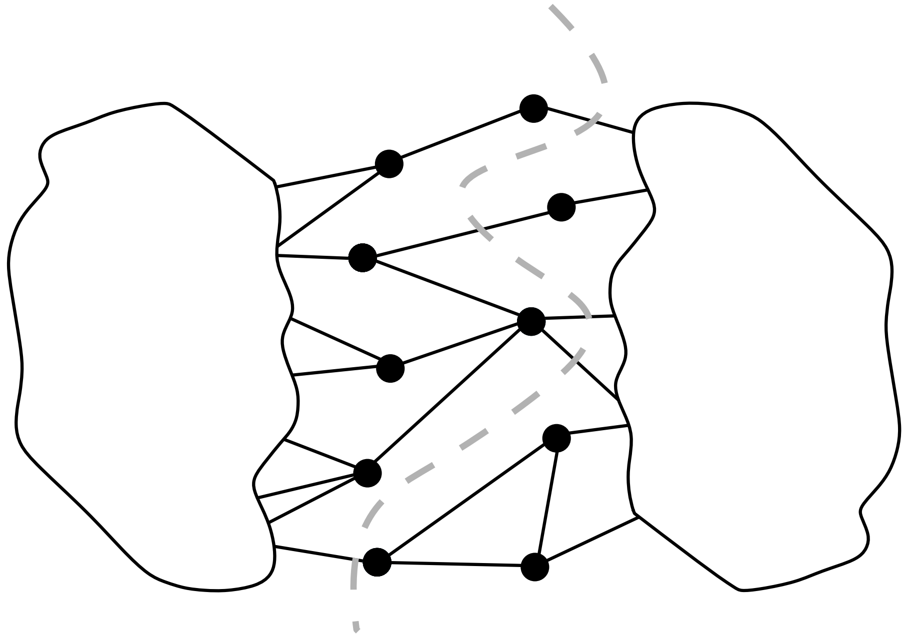
Fiduccia-Mattheyses Algorithm

FM Local Search

while ¬ done **do**
    find best move
    perform best move
rollback to best solution



can worsen solution



- recalculate gain $g(v)$ of neighbors
- move each node at most once
- edge-cut: **7, 6,5,5,6**

rollback

Institute of Theoretical Informatics
Algorithmics Group

Parallelization

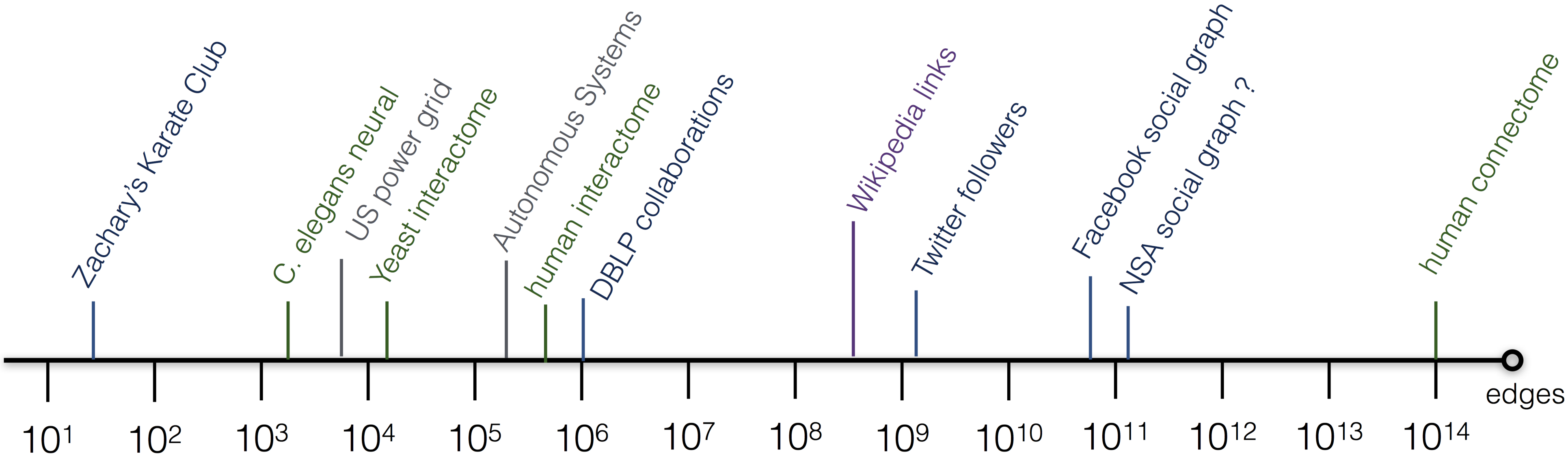
All presented problems have parallel algorithms:

- some problems are well suited for parallelization
  - BFS algorithms – especially trees, DAGs
  - MST algorithms – local cut or cycle property
- if global decisions are required for exact solutions
  - less suitable for parallel processing
  - e.g. coloring, independent sets, . . .
  - often parallelizable greedy heuristics $\Rightarrow$ only need local criteria

Network Analysis

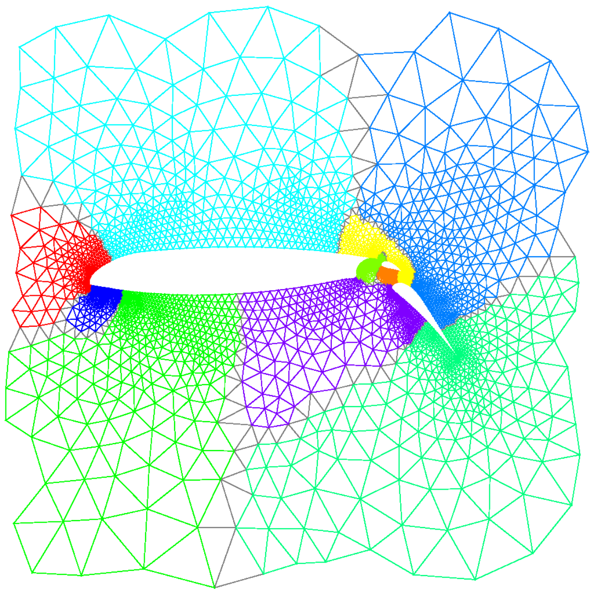
Network Analysis

- Transportation
- Business
- (Online) Social networks
- Technology
- Biology

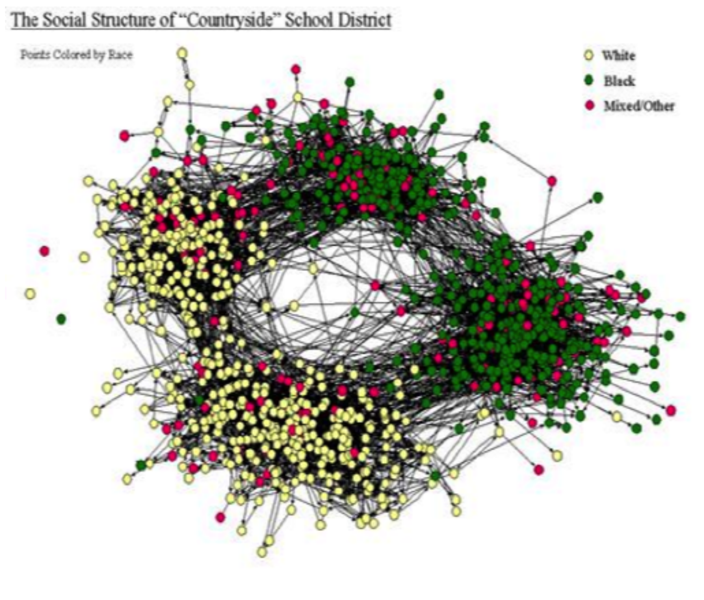


Complex Networks

- Non-trivial **topological features** that do not occur in simple networks (meshes, simple random graphs), but often occur in reality
  - Small diameter
  - Strongly varying degree distribution
  - Large number of triangles
  - ...



Airfoil mesh



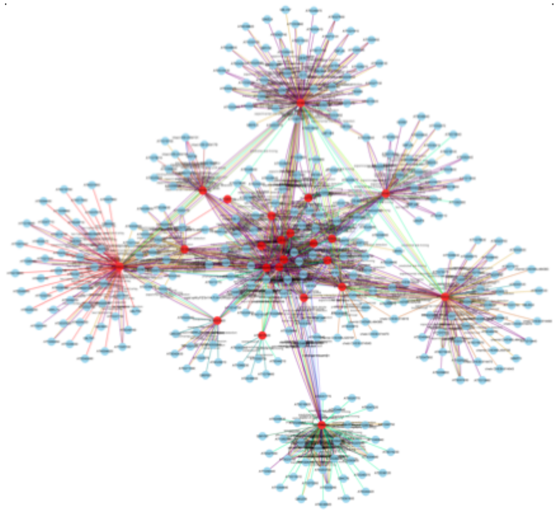
Social network

Institute of Theoretical Informatics
Algorithmics Group

Example Applications

Bioinformatics

- **Protein-protein interactions**
- Phylogeny trees
- ...

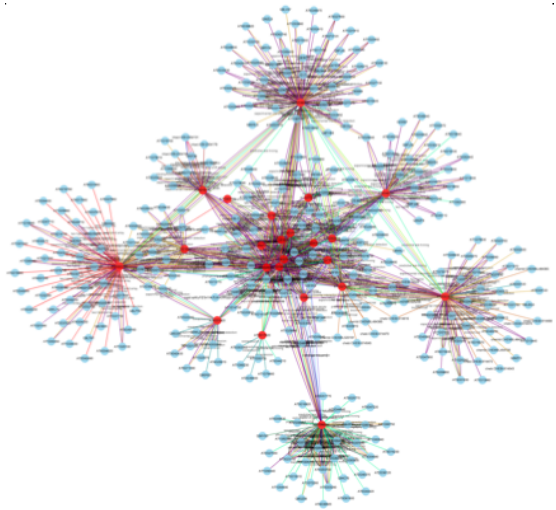


PPI network

Example Applications

Bioinformatics

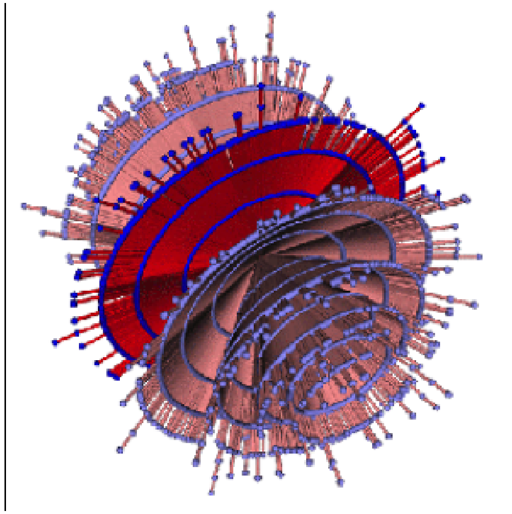
- **Protein-protein interactions**
- Phylogeny trees
- ...



PPI network

Collaborations

- **Movies**
- Scientific papers
- Politics
- ...



Six degrees of Kevin Bacon

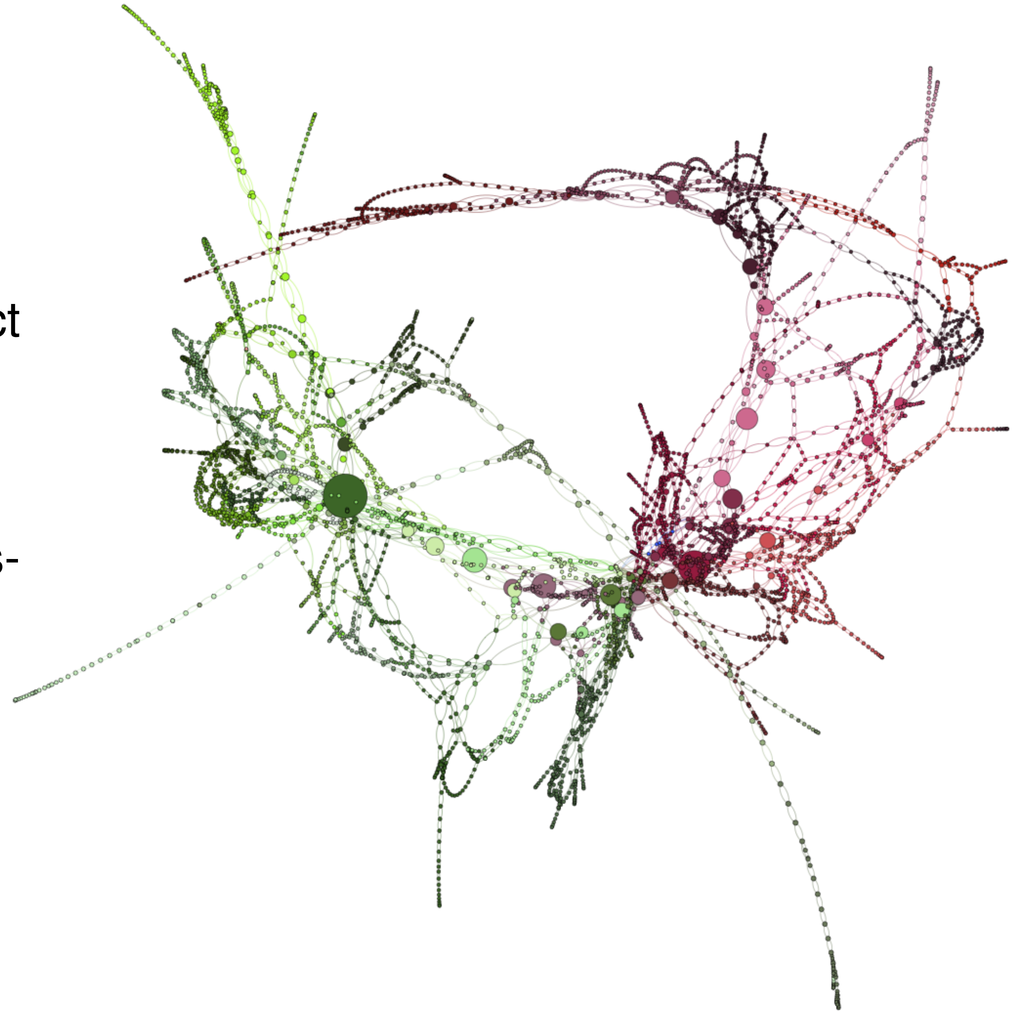
[Seok-Hee Hong]

Network Science

"Statistics of relational data"

Often

- exploratory in nature
- requires data preprocessing to extract graph
- creates large data sets easily
- requires domain-specific postprocessing for interpretation



[sayasaya2011.wordpress.com/]

NetworKit

NetworKit: parallel tool suite for network analysis

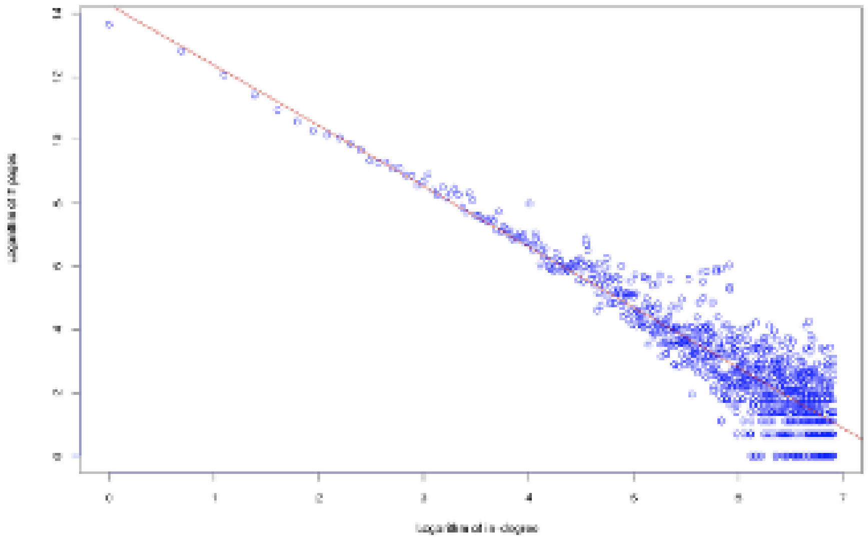
- large collection of network science algorithms
- shared-memory parallel C++ implementation
- Python interface
- suitable for interactive analysis with IPython notebooks

For all introduced measures: NetworKit IPython call

Degree Distribution

Concept

- Interesting: Distribution of node degrees
- Typically heavy-tailed
  (especially power law $p(k) \sim k^{-\gamma}$)
- Example: Web graphs



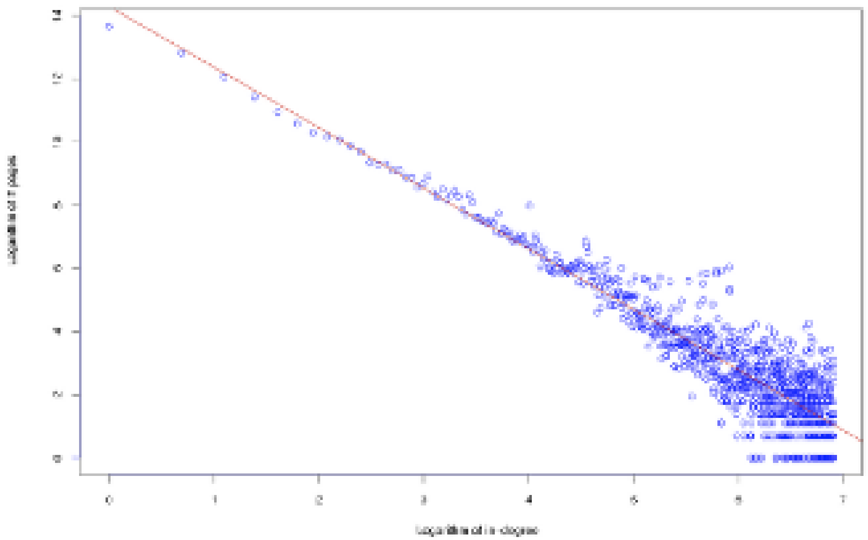
[http://www2002.org/CDROM/poster/164/]

Graph of African web pages early 2000s

Degree Distribution

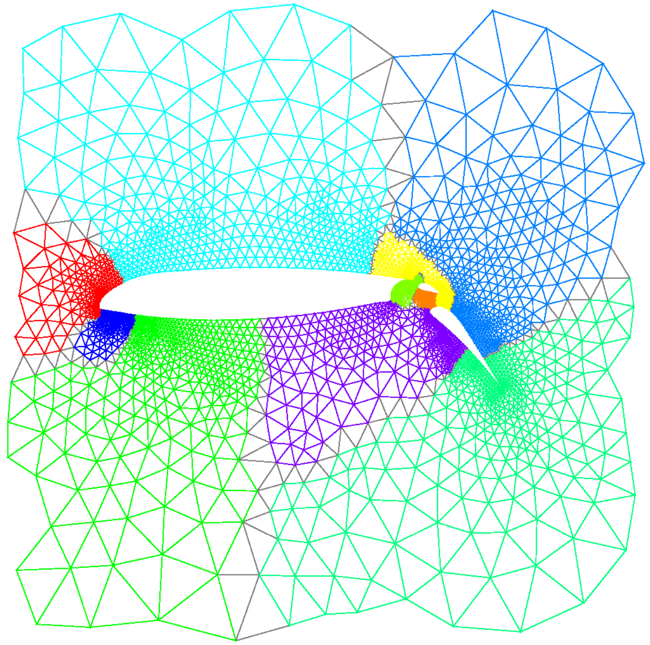
Concept

- Interesting: Distribution of node degrees
- Typically heavy-tailed
  (especially power law $p(k) \sim k^{-\gamma}$)
- Example: Web graphs



[http://www2002.org/CDROM/poster/164/]

Graph of African web pages early 2000s



Not heavy tailed, often constant: Meshes

[Clauset et al. 2009: Power-law distributions in empirical data]

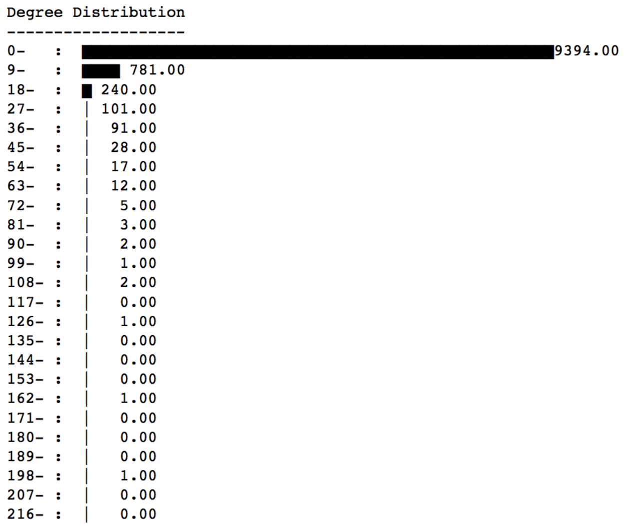
Degree Distribution

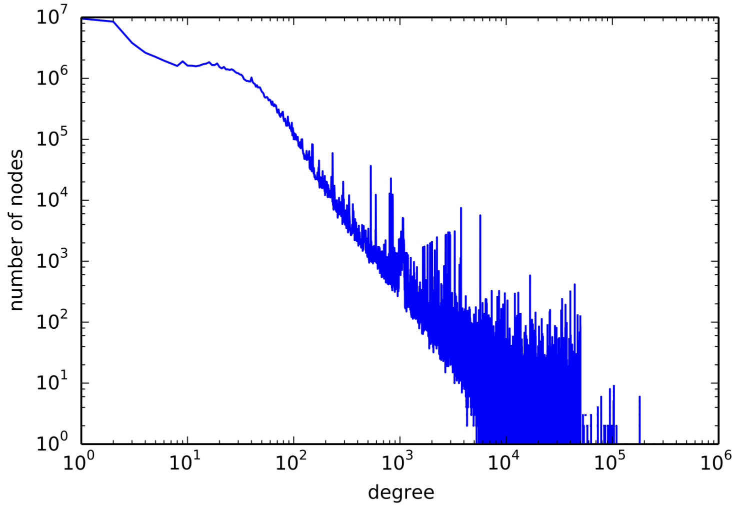
Algorithms

- Visualizations of degree distribution
- `powerlaw` Python module determines whether distribution fits power law and estimates exponent γ

Good: $O(|E|)$

```
dd = centrality.DegreeCentrality(G)
```



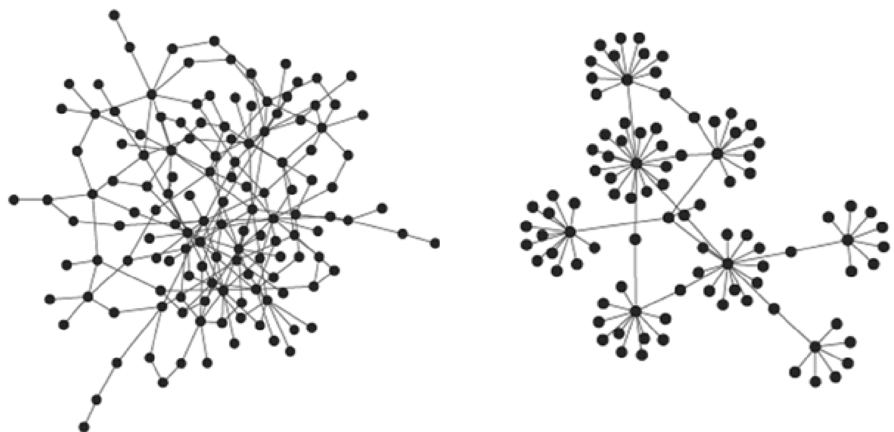


[Alstott et al. 2014: powerlaw: a python package for analysis of heavy-tailed distributions.]

Degree Assortativity

Concept

- Formation of connections between nodes with similar/dissimilar degree
- Based on covariance of degrees
- Normalization expressed as correlation coefficient r
- Let $k_i := d(i)$:



$$\mathrm{cov}(k_i, k_j) = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \frac{k_i k_j}{2m} \right) k_i k_j$$

$$r = \frac{\sum_{i,j}(A_{ij} - k_i k_j / 2m) k_i k_j}{\sum_{i,j}(k_i \delta_{ij} - k_i k_j / 2m) k_i k_j} \qquad \delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

[Newman: Networks – An Introduction. Chapters 7.13, 8.7] [Newman 2002: Assortative mixing in networks.]

Degree Assortativity

Algorithm

- Original formula disadvantageous for computation

$$r = \frac{\sum_{i,j}(A_{ij} - k_i k_j/2m)k_i k_j}{\sum_{i,j}(k_i \delta_{ij} - k_i k_j/2m)k_i k_j} \qquad \delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

- Reformulation (see Newman):

$$r = \frac{S_1 S_e - S_2^2}{S_1 S_3 - S_2^2} \qquad S_e = \sum_{i,j} A_{ij} k_i k_j = 2 \sum_{\{i,j\} \in E} k_i k_j$$

$$S_1 = \sum_i k_i \qquad S_2 = \sum_i k_i^2 \qquad S_3 = \sum_i k_i^3$$

```
da = correlation.Assortativity(G, dd)
```

Good: $O(|E|)$

Institute of Theoretical Informatics
Algorithmics Group

k-Core Decomposition

Concept

- Nodes in core k have at least k neighbors that also belong to core k, $k \geq 0$
- Iteratively peeling away nodes of degree k reveals the k-cores

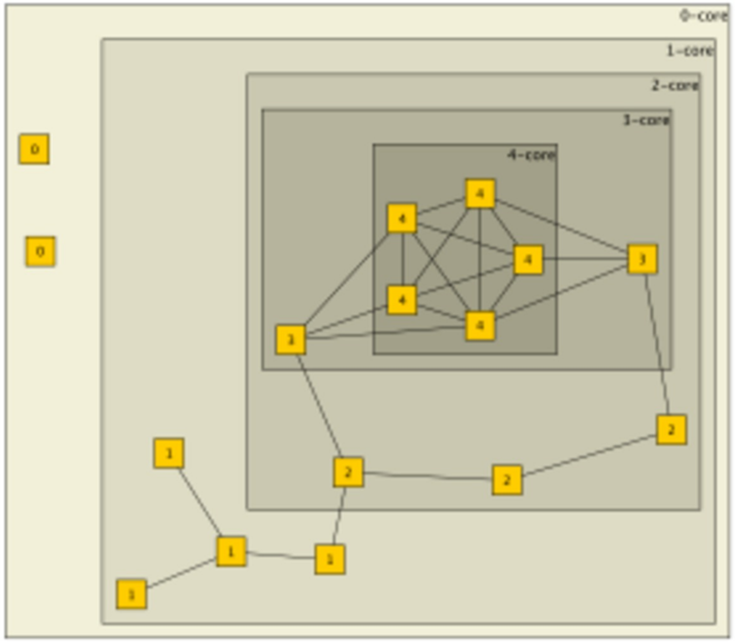


Fig. 1. A k-core decomposition with 5 core shells.

[Baur et al. 2008]

k-Core Decomposition

Karlsruhe Institute of Technology

Concept

- Nodes in core k have at least k neighbors that also belong to core k, $k \geq 0$
- Iteratively peeling away nodes of degree k reveals the k-cores

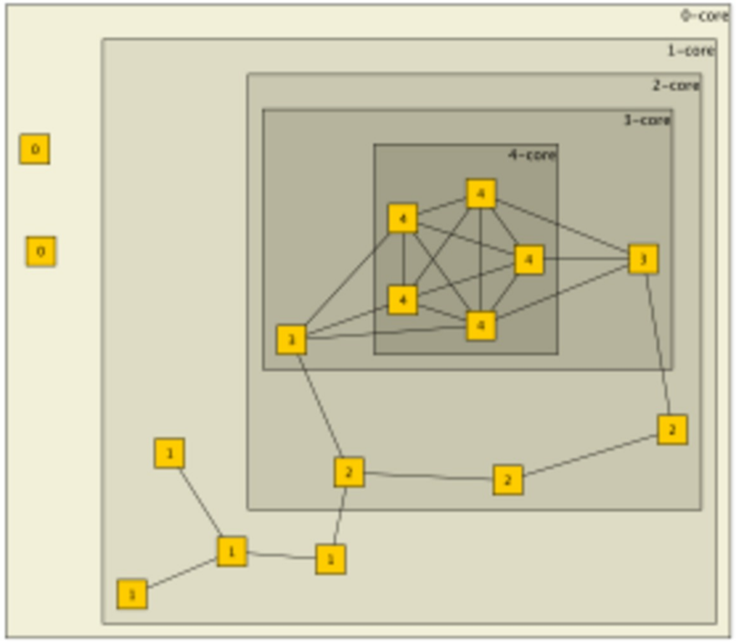


Fig. 1. A k-core decomposition with 5 core shells.

[Baur et al. 2008]

store node degrees in array `degree`
$i \leftarrow 1$
while $V \neq \emptyset$ **do**
  for each $v \in V$ with `degree`$[v] < i$ **do**
    ... $\triangleright$ process v and its neighbors and delete v from G
  $i \leftarrow i + 1$
return (i-1, `core`)

k-Core Decomposition

Algorithm and Implementation

- Bucket data structure
- Each bucket stores nodes with the same current degree
- Additional array to store pointers from each node into its bucket

1: **for** each $v \in V$ with $\texttt{degree}[v] < i$ **do**
2: $\quad \texttt{core}[v] \leftarrow i - 1$
3: $\quad$ **for** each $u \in N(v)$ **do**
4: $\quad\quad \texttt{degree}[u] \leftarrow \texttt{degree}[u] - 1$
5: $\quad$ Remove v from G

```
coreDec = centrality.CoreDecomposition(G)
```

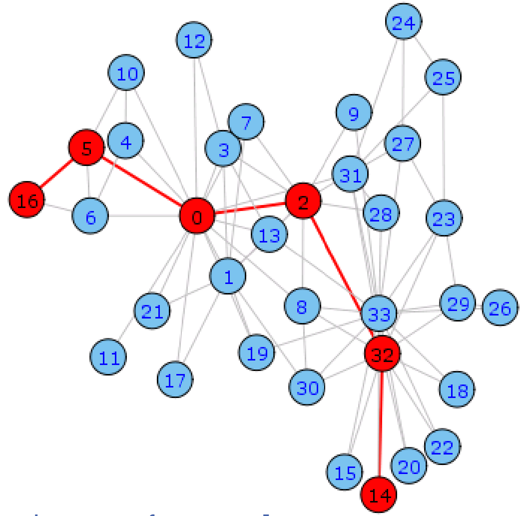
Good: $O(|E|)$

Institute of Theoretical Informatics
Algorithmics Group

Diameter

Concept

- Longest shortest path between any two nodes
- Small in most complex networks
- “Six degrees of separation”



[igraph.sourceforge.net]

Algorithms

- Exact: Simple all pairs shortest paths
  (n shortest path queries)
- In practice faster: iFub
- $\frac{3}{2}$-approximation possible in $O\left(|E|\sqrt{|V|}\right)$

`diam = distance.Diameter(G)`

Goodish: $O\left(|V|\,|E|\right)$

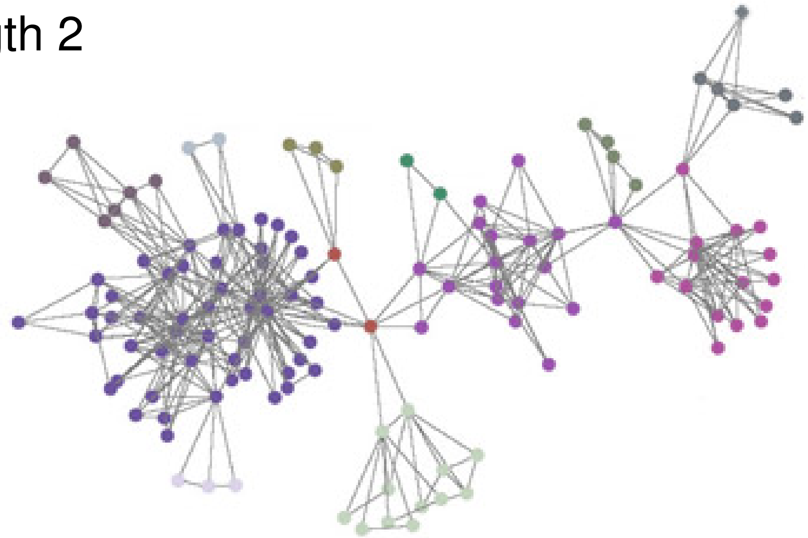
[Crescenzi et al. 2013: On computing the diameter of real-world undirected graphs]

[Roditty, Williams. 2013: Fast Approx. Algorithms for the Diameter and Radius of Sparse Graphs]

Clustering Coefficients

Concept

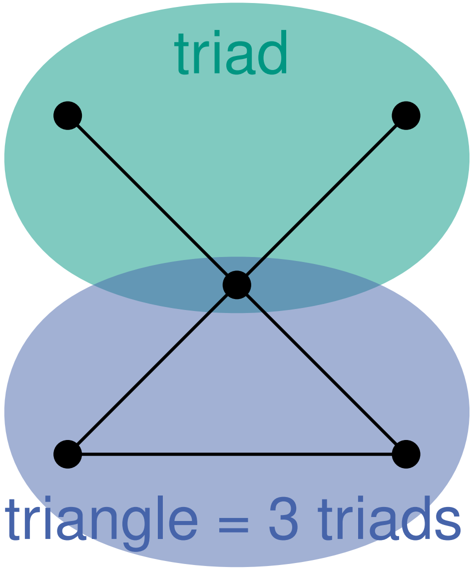
- Social networks: High ratio of closed triangles
  (“Friends of friends are often friends”)
- CC: Ratio of closed triangles and paths of length 2

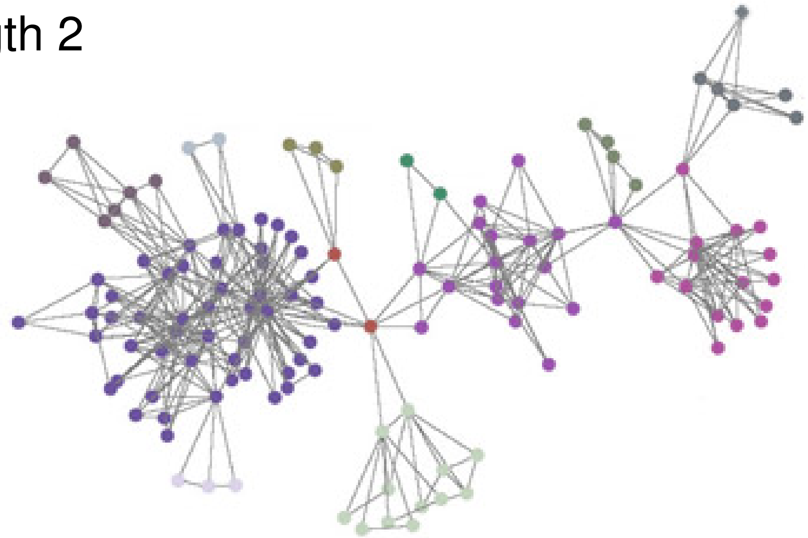


Clustering Coefficients

Concept

- Social networks: High ratio of closed triangles
  ("Friends of friends are often friends")
- CC: Ratio of closed triangles and paths of length 2





$$C_g(G) = \frac{3 \cdot \text{Number of closed triangles}}{\text{Number of connected triads}}$$

$$C_l(v) = \frac{\text{Number of triangles with } v}{\text{Number of connected triads with } v \text{ as middle node}}$$

Clustering Coefficients

Exact Algorithm

- with parallel node iteration: $O(|V|\, d_{\max}^2)$ time

Approximation

- **Wedge sampling:**
  Linear-time approximation for weighted graphs with probabilistic absolute error ϵ

```
cc = globals.ClusteringCoefficient(G)
```

Good: $O(|E|)$

[Schank, Wagner 2005: Approximating clustering coefficient and transitivity]

Centrality Measures

Centrality Concept

- How important is a node / an edge?

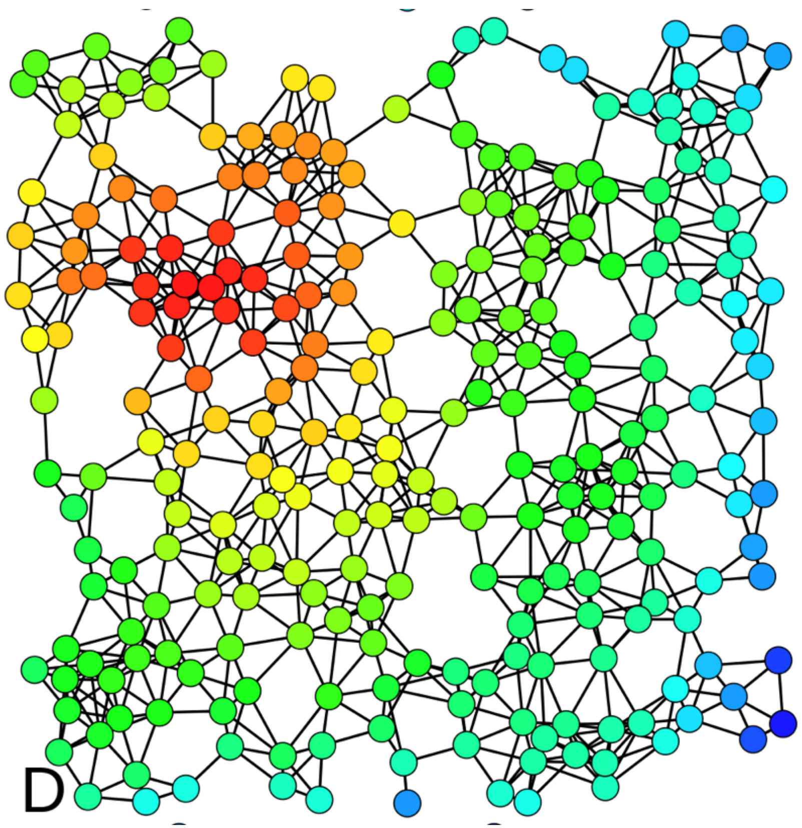
Eigenvector Centrality

- Consider importance of neighbors:

$$\forall v \in V : x_v = \frac{1}{\lambda} \sum_{u \in V} A_{vu} x_u$$

$$\lambda \mathbf{x} = A\mathbf{x}$$ $A :=$ adjacency matrix

- Eigenvector to largest eigenvalue




```
ec = centrality.EigenvectorCentrality(G)
```

Goodish: $O\left(|V|^3\right)$

Institute of Theoretical Informatics
Algorithmics Group

Centrality Measures

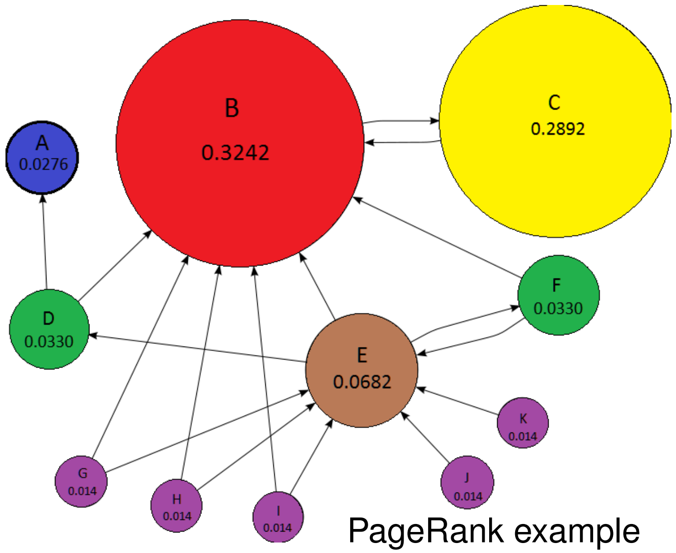
Centrality Concept

- How important is a node / an edge?

PageRank

- Google's first ranking scheme
- variant of eigenvector centrality
- Random surfer model:

$$\forall v \in V : x_v^{(t+1)} = \alpha \cdot \frac{1}{|V|} + (1-\alpha) \sum_{(u \mapsto v) \in E} \frac{x_u^{(t)}}{|\{(u \mapsto x) \in E\}|}$$



PageRank example

```
ec = centrality.PageRank(G, 1e-6)
```

Goodish: $O\left(|V|^3\right)$

Institute of Theoretical Informatics
Algorithmics Group

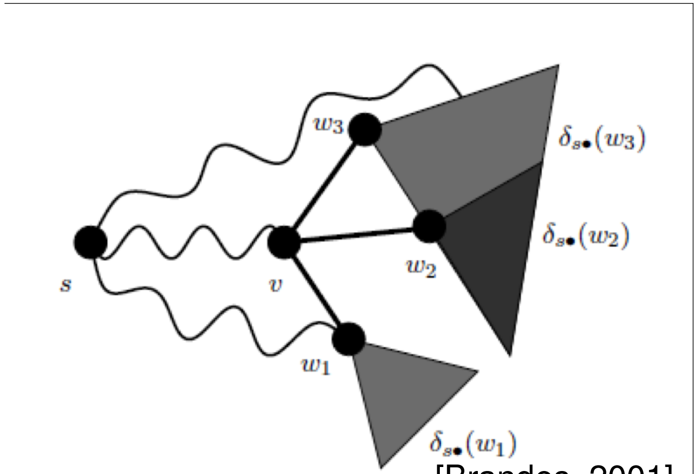
Betweenness Centrality

Definition

- $\forall u, v \in V$ in connected graph, there exists at least one shortest path between them
- BC measures of number of shortest paths that pass through a vertex k

$$C_B(k) = \sum_{u,v \in V \setminus \{k\}} \frac{|\{k \in SP(u,v)|\}}{|SP(u,v)|}$$

$SP(u, v)$ = shortest paths from u to v



[Brandes, 2001]

Betweenness Centrality

Definition

- $\forall u, v \in V$ in connected graph, there exists at least one shortest path between them
- BC measures of number of shortest paths that pass through a vertex k

$$C_B(k) = \sum_{u,v \in V \setminus \{k\}} \frac{|\{k \in SP(u,v)|\}}{|SP(u,v)|}$$

$SP(u, v)$ = shortest paths from u to v

Exact Algorithm for BC

- Brandes's alg.: $O(|V|\,|E| + |V|^2 \log |V|)$ time

Approximation for BC

- Parallel path sampling with probabilistic absolute error (in (nearly-)linear time)

```
bc = centrality.Betweenness(G)
```

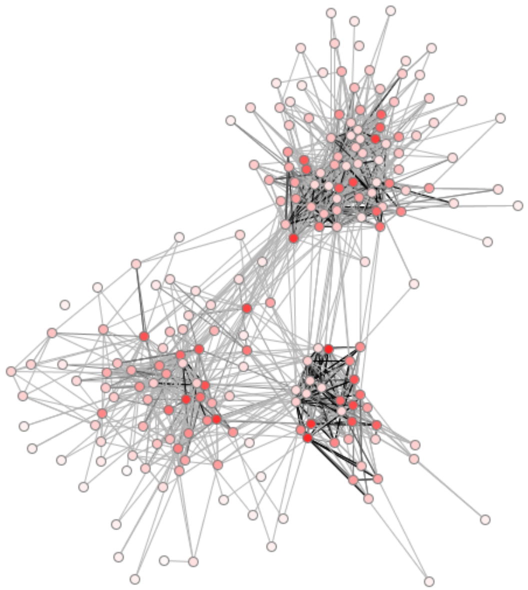
Goodish: $\tilde{O}\left(|V|^3\right)$

[Brandes 2001: A faster algorithm for betweenness centrality]
[Riondato, Kornaropoulos 2013: Fast approximation of betweenness centrality through sampling]
[Geisberger et al. 2008: Better Approximation of Betweenness Centrality]

Community Detection (CD)

Community Detection / Graph Clustering

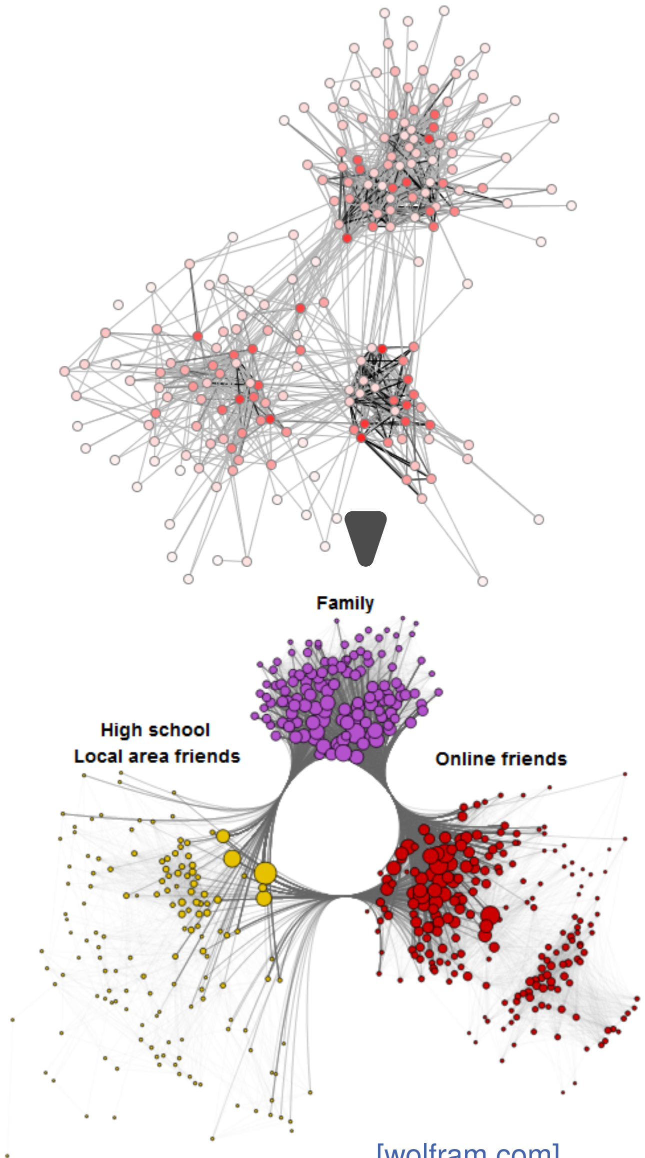
- Find (non-overlapping) **internally dense, externally sparse subgraphs**
- Goals: Uncover community structure, prepartition network
- number of cluster not known in advance ⇔ partitioning



Community Detection (CD)

Community Detection / Graph Clustering

- Find (non-overlapping) **internally dense, externally sparse subgraphs**
- Goals: Uncover community structure, prepartition network
- number of cluster not known in advance ⇔ partitioning



[wolfram.com]

What constitutes a cluster?

[survey: Schaeffer 07, Fortunato 10]

[Girvan, Newman 2002: Community structure in social and biological networks]

CD – Objective Functions

Given a clustering $\mathcal{C}$ for a graph G:

- **Coverage:** fraction of intra-cluster edges $\omega(\mathcal{C})$ over all edges

$$\text{cov}(\mathcal{C}) := \frac{\omega(\mathcal{C})}{|E|}$$

- Problem: maximal for trivial cluster ($k = 1$)

Bad: NP-hard

CD – Objective Functions

Given a clustering $\mathcal{C}$ for a graph G:

- **Coverage:** fraction of intra-cluster edges $\omega(\mathcal{C})$ over all edges

$$\operatorname{cov}(\mathcal{C}) := \frac{\omega(\mathcal{C})}{|E|}$$

- Problem: maximal for trivial cluster ($k = 1$)

Bad: NP-hard

- **Performance:** fraction node pairs that are clustered correctly

$$\operatorname{perf}(\mathcal{C}) := \frac{m(\mathcal{C}) + \bar{m}^c(\mathcal{C})}{\frac{1}{2}|V|(|V| - 1)}$$

$$m(\mathcal{C}) := |\{(u, v) \in E : \mathcal{C}(u) = \mathcal{C}(v)\}|$$

$$\bar{m}^c(\mathcal{C}) := |\{u, v \in V : \mathcal{C}(u) \neq \mathcal{C}(v) \ \& \ (u, v) \notin E\}|$$

- Problem: in sparse networks $\bar{m}^c(\mathcal{C})$ dominates $\Rightarrow$ fine clusterings

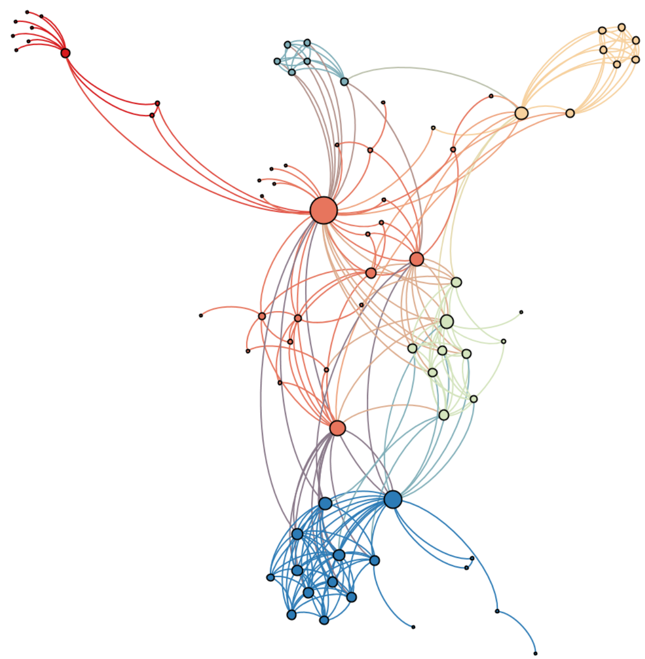
Bad: NP-hard

CD – Objective Functions

- **Modularity:** cov(·) minus expected coverage of random graph with same clustering

$$\text{mod}(\mathcal{C}) = \text{cov}(\mathcal{C}) - \mathbb{E}\left[\text{cov}(\mathcal{C})\right]$$

$$= \frac{\omega(\mathcal{C})}{|E|} - \frac{1}{4\,|E|^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} d(v) \right)^2$$



CD – Objective Functions

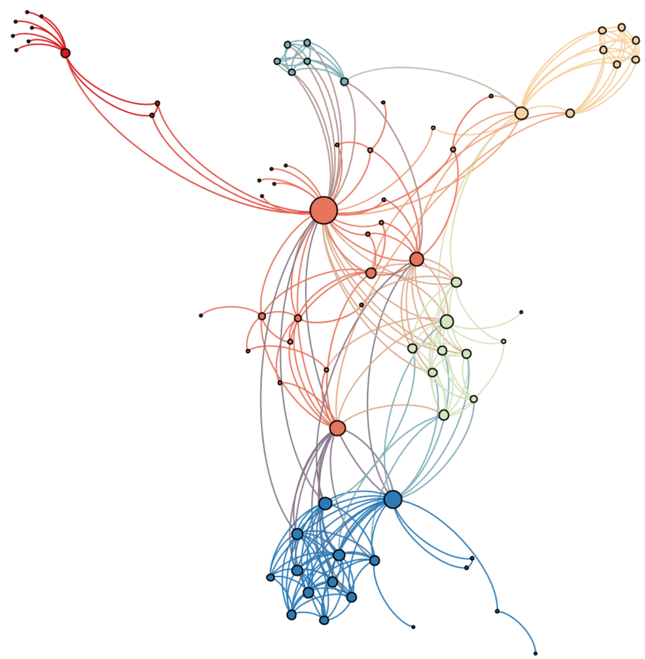
- **Modularity:** cov(·) minus expected coverage of random graph with same clustering

$$\mathrm{mod}(\mathcal{C}) = \mathrm{cov}(\mathcal{C}) - \mathbb{E}\left[\mathrm{cov}(\mathcal{C})\right]$$

$$= \frac{\omega(\mathcal{C})}{|E|} - \frac{1}{4\,|E|^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} d(v) \right)^2$$

favors many edges in cluster

favors many clusters with small degree



Institute of Theoretical Informatics
Algorithmics Group

CD – Objective Functions

- **Modularity:** cov(·) minus expected coverage of random graph with same clustering

$$\mathrm{mod}(\mathcal{C}) = \mathrm{cov}(\mathcal{C}) - \mathbb{E}\left[\mathrm{cov}(\mathcal{C})\right]$$

$$= \frac{\omega(\mathcal{C})}{|E|} - \frac{1}{4|E|^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} d(v) \right)^2$$

favors many edges in cluster

favors many clusters with small degree

- random graph with same degree distribution
- agrees well with intuitive clustering of graph
- Modularity has some **known issues** (resolution limit, ...), some can be circumvented
- most popular clustering metric in network analysis

Ugly: NP-hard, not APX

[Brandes et al. 2006: On Modularity – NP-Completeness and Beyond]
[Dinh et al. 2016: Network Clustering via Maximizing Modularity: Approximation Algorithms and Theoretical Limits]

CD – Algorithms

But in practice well-functioning algorithms available:

- parallel label propagation (PLP)
- parallel Louvain method (PLM)
- PLM with refinement (PLMR)

```
cd = community.detectCommunities(G)
```

Good: $O\left(|V| \log |V|\right)$

Institute of Theoretical Informatics
Algorithmics Group

CD – Algorithms

But in practice well-functioning algorithms available:

- parallel label propagation (PLP)
- parallel Louvain method (PLM)
- PLM with refinement (PLMR)

```
cd = community.detectCommunities(G)
```

Good: $O\left(|V| \log |V|\right)$

Louvain Method: two-phase iterative algorithm

- place each node in their own cluster

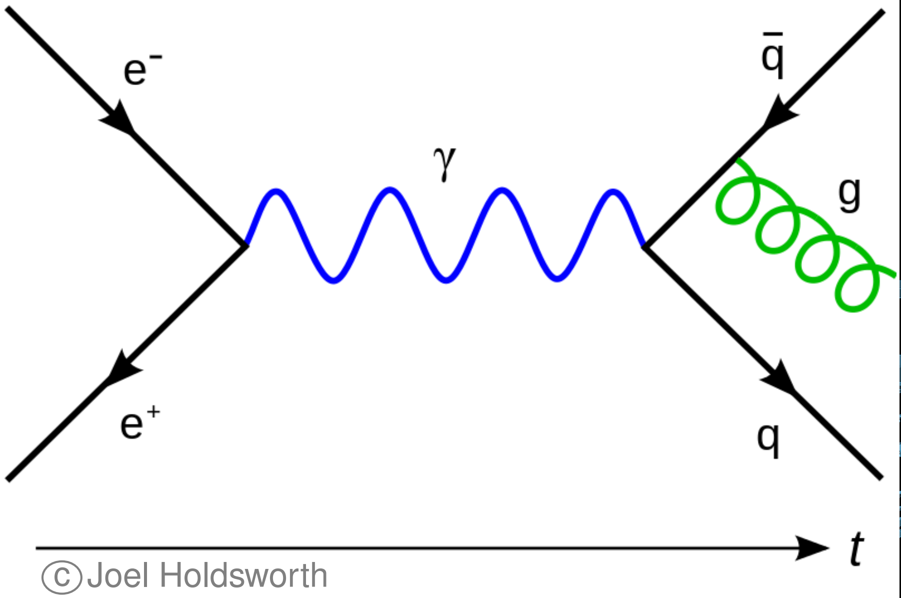
1. $\forall v$: calculate $\Delta\,\mathrm{mod}(\cdot)$ for moving v to any of its neighboring clusters
   - perform most effective move
   - repeat until no more gain possible
2. contract all clusters to one node
   - intra-cluster edges become self loops
   - inter-cluster edges represented by weighted edges

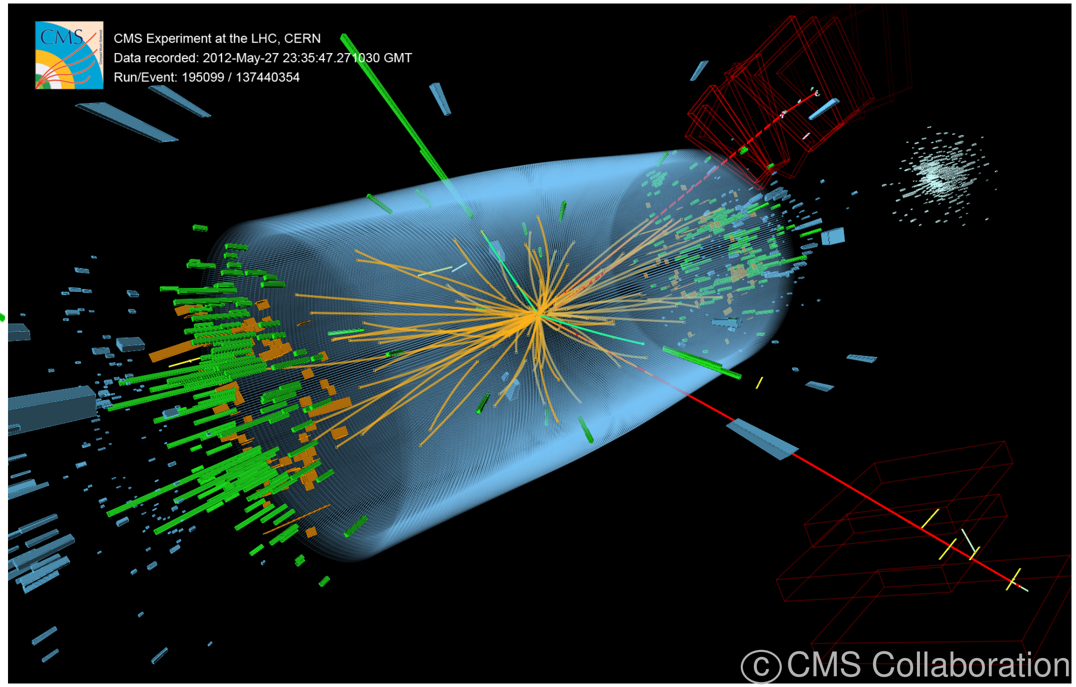
Case Studies in Physics

Case Studies in Physics

Graphs can be applied in varied areas of physics

- graphs to gain theoretical insight: Feynman diagrams
- graphs to model physical problems: particle track reconstruction
- graphs to speed up an algorithms: jet clustering





Institute of Theoretical Informatics
Algorithmics Group

Theoretical Applications

- graph coloring can be applied to Feynman Diagrams to determine the presence of particular Feynman integrals

Theoretical Applications

- graph coloring can be applied to Feynman Diagrams to determine the presence of particular Feynman integrals

The ϕ^k theory is compared with the multilinear theory of scalar fields $\phi_1, \phi_2, \ldots, \phi_k$ having the same mass as that of ϕ. In particular, it is shown that Feynman integrals encountered in the ϕ^3 theory are not necessarily present also in the ϕ_1, ϕ_2, ϕ_3 theory, but they are if they correspond to planar Feynman graphs having no tadpole part. Furthermore, a necessary and sufficient condition for the presence of a ϕ^3 Feynman integral in the ϕ_1, ϕ_2^2 theory is found. Those considerations are applications of graph theory, especially of the coloring problem of graphs, to Feynman graphs.

[Nakanishi, Noboru. Quantum field theory and the coloring problem of graphs. Comm. Math. Phys. 32 (1973), no. 2, 167–181.]

Theoretical Applications

- graph coloring can be applied to Feynman Diagrams to determine the presence of particular Feynman integrals

The ϕ^k theory is compared with the multilinear theory of scalar fields $\phi_1, \phi_2, \ldots, \phi_k$ having the same mass as that of ϕ. In particular, it is shown that Feynman integrals encountered in the ϕ^3 theory are not necessarily present also in the ϕ_1, ϕ_2, ϕ_3 the… n graphs having no tadpole part. Furthermore, a necessary … man integral in the ϕ_1, ϕ_2^2 theory is found. Those considerations are applications of graph theory, especially of the coloring problem of graphs, to Feynman graphs.

Beyond physics understanding of three computer scientists

[Nakanishi, Noboru. Quantum field theory and the coloring problem of graphs. Comm. Math. Phys. 32 (1973), no. 2, 167–181.]

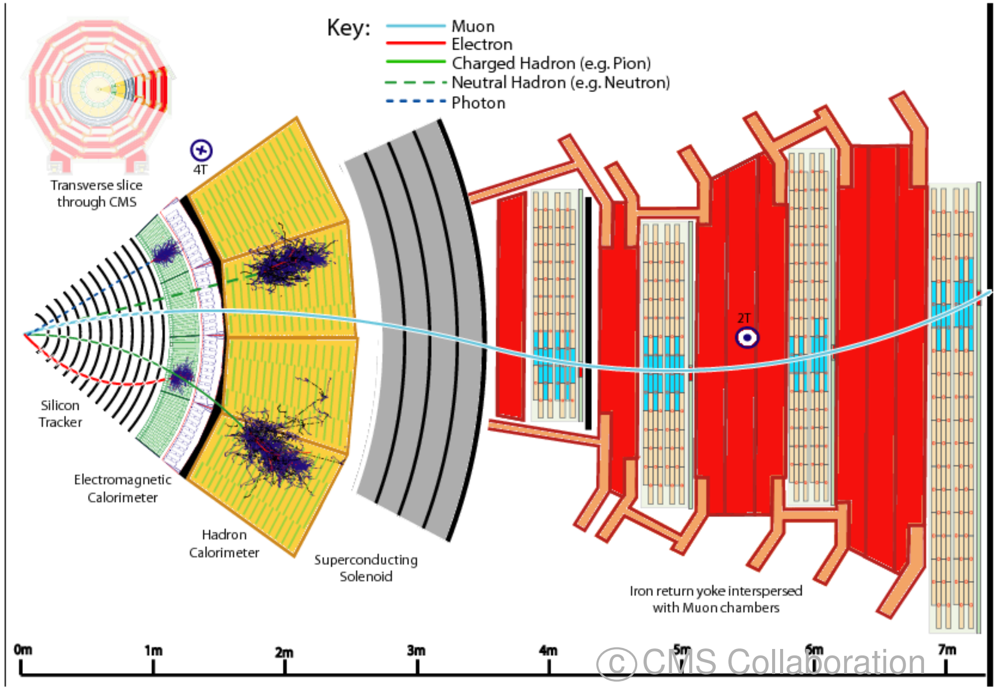
Theoretical Applications

- graph coloring can be applied to Feynman Diagrams to determine the presence of particular Feynman integrals
- further results in condensed matter physics, statistical physics,. . .

  [Estrada, E. (2013): Graph and Network Theory in Physics, ArXiv 1302.4378]

Particle Track Reconstruction

- particles traverse several multi-layer detectors after collision
  ⇒ particularly inner tracker
- energy deposits in detector material are reconstructed as hits
- particle track reconstruction ⇒ combinatoral pattern matching problem

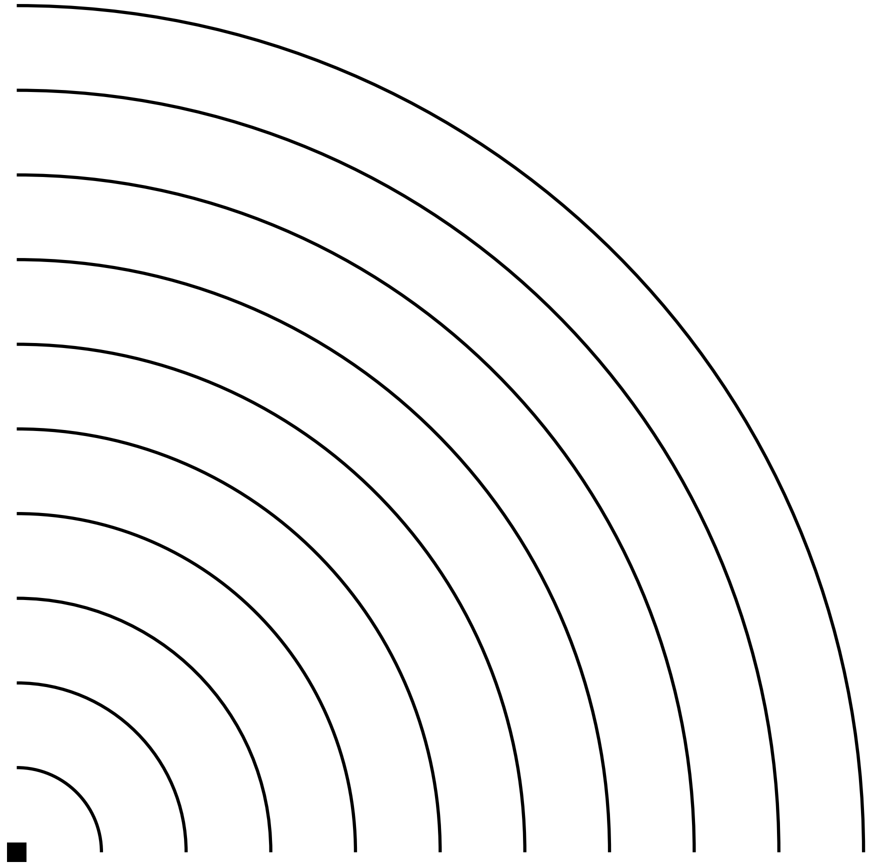


Institute of Theoretical Informatics
Algorithmics Group

Particle Track Reconstruction

Approach most used: Iterative Kalman Filter Track Finding

1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing

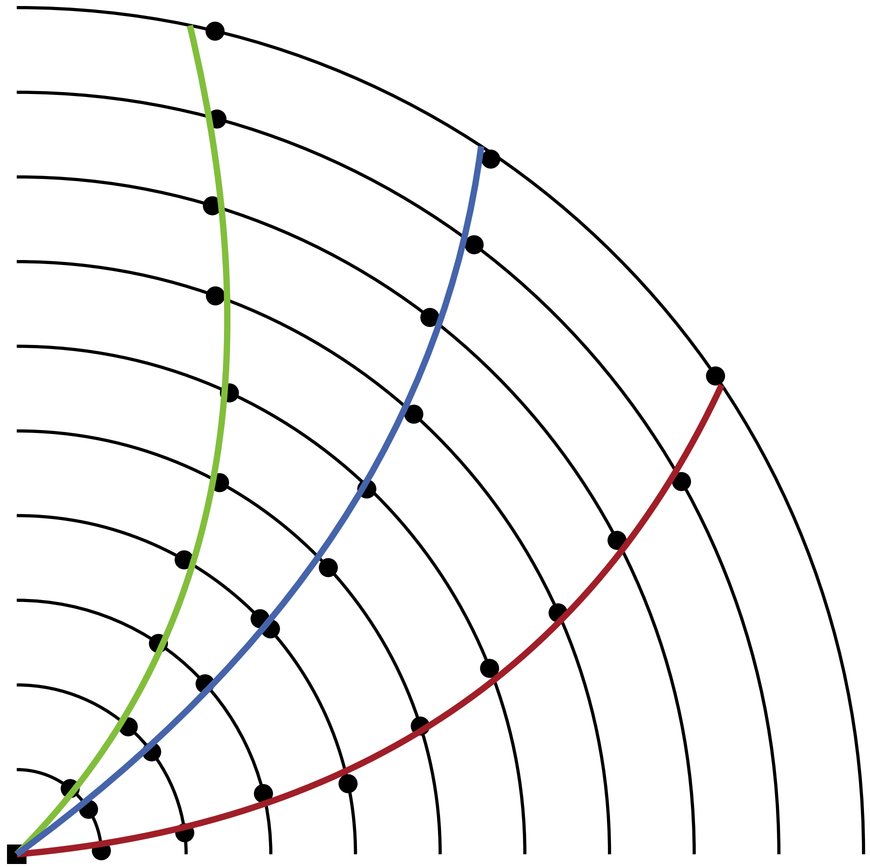


Institute of Theoretical Informatics
Algorithmics Group

Particle Track Reconstruction

Approach most used: Iterative Kalman Filter Track Finding

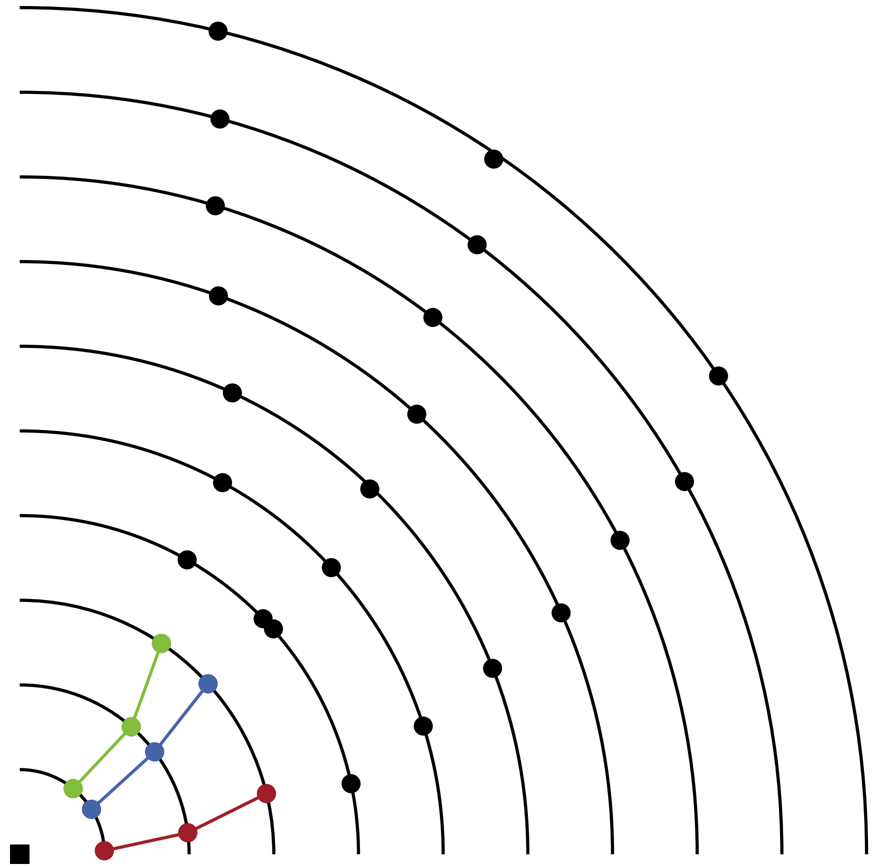
1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing



Particle Track Reconstruction

Approach most used: Iterative Kalman Filter Track Finding

1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing

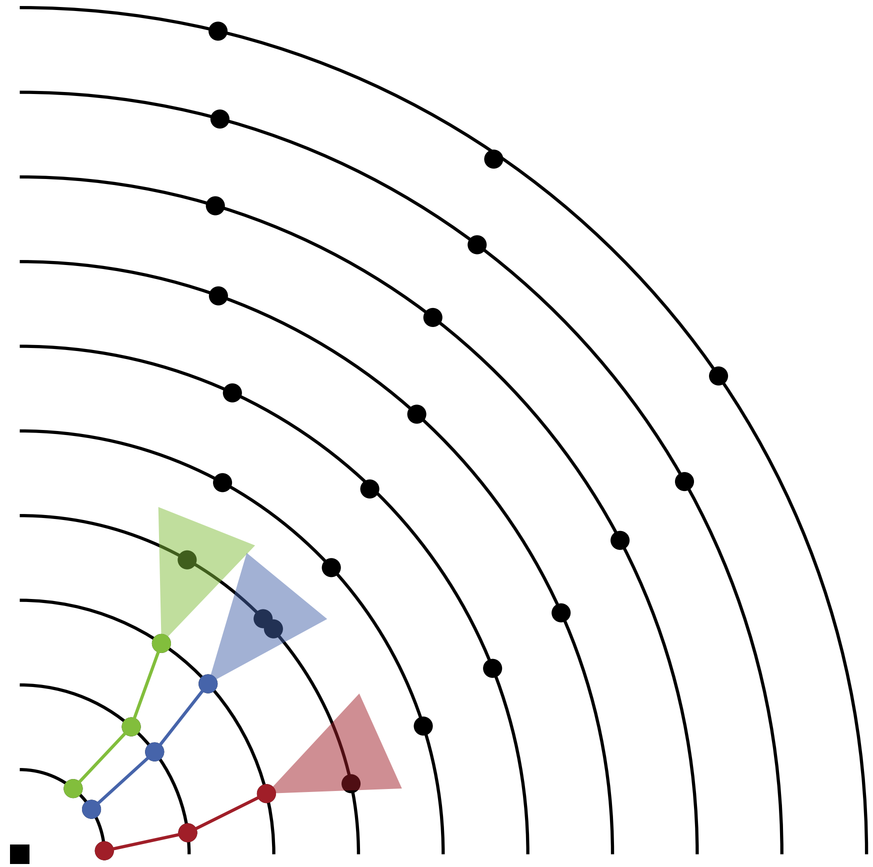


Institute of Theoretical Informatics
Algorithmics Group

Particle Track Reconstruction

Approach most used: Iterative Kalman Filter Track Finding

1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing

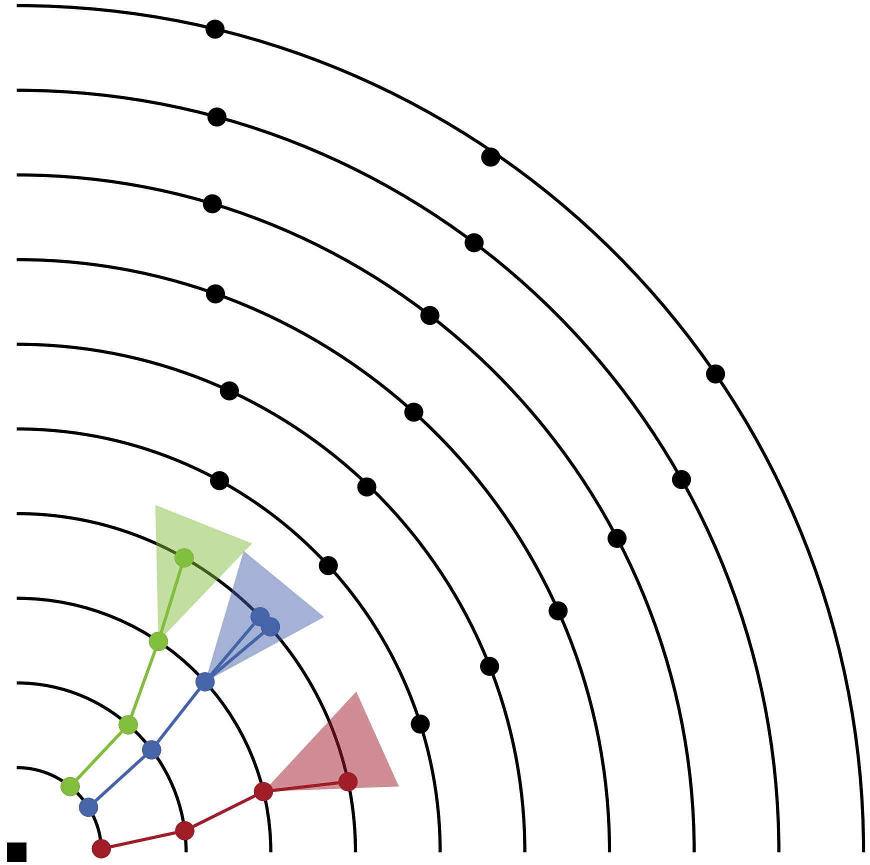


Institute of Theoretical Informatics
Algorithmics Group

Particle Track Reconstruction

Approach most used: Iterative Kalman Filter Track Finding

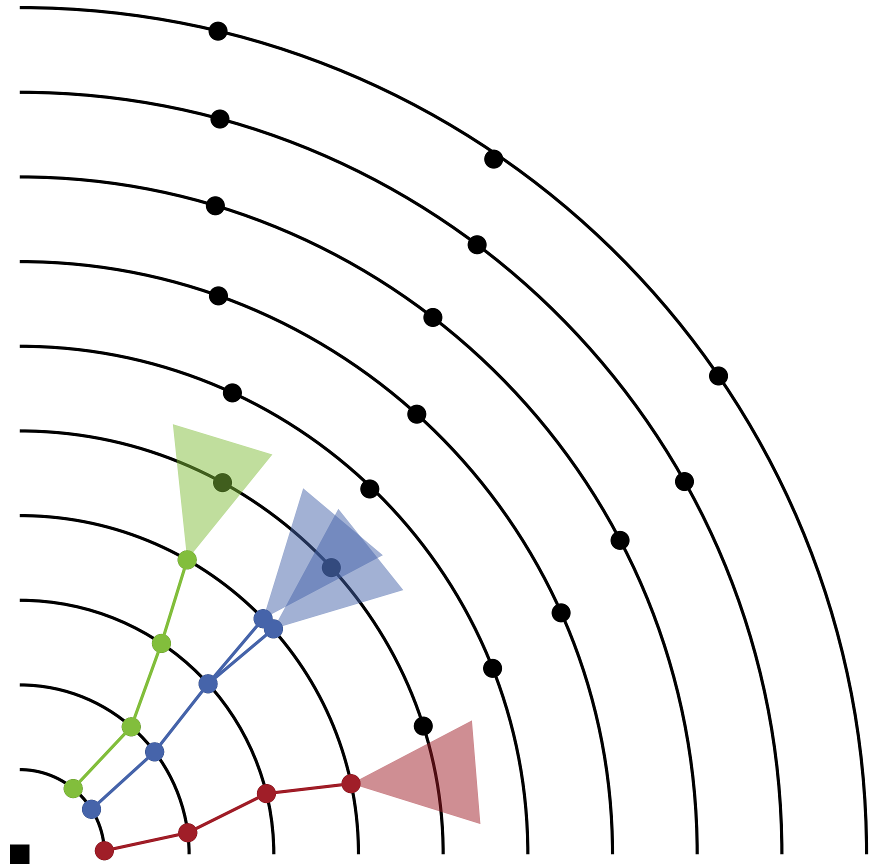
1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing



Particle Track Reconstruction

Approach most used: Iterative Kalman Filter Track Finding

1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing

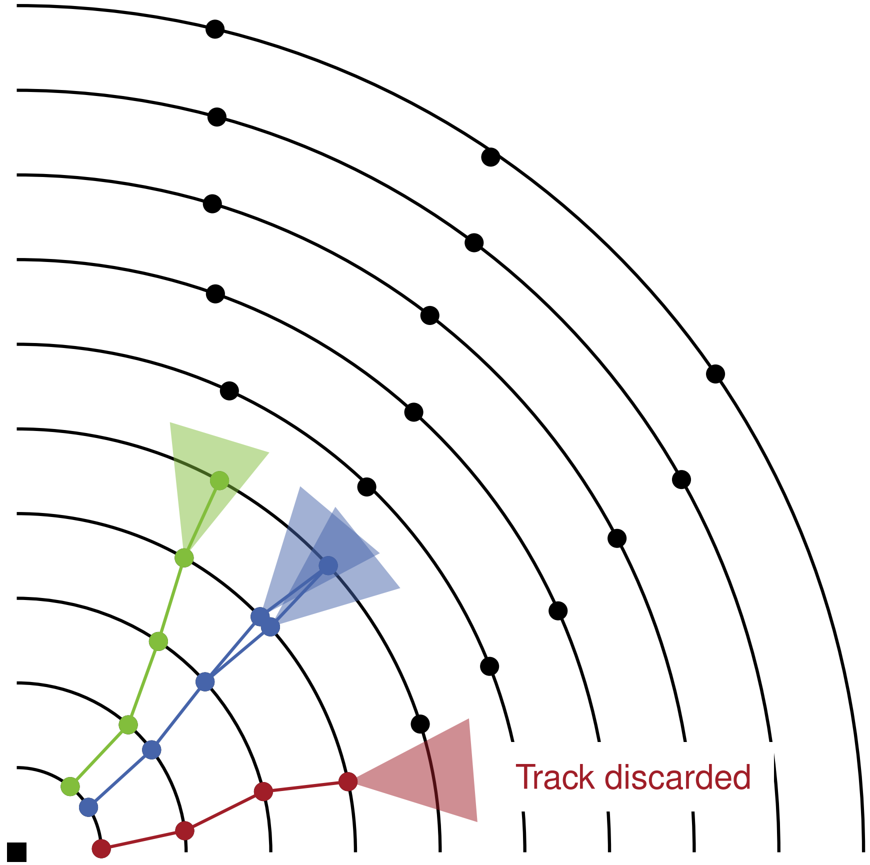


Particle Track Reconstruction

Karlsruhe Institute of Technology

Approach most used: Iterative Kalman Filter Track Finding

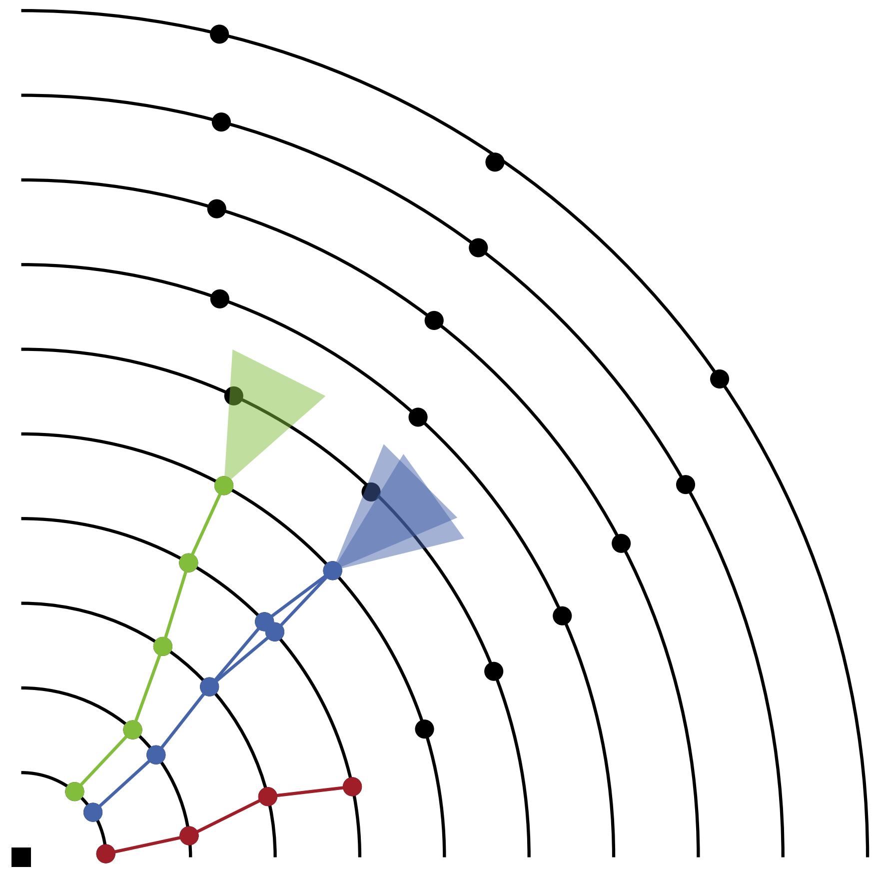
1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing



Particle Track Reconstruction

Approach most used: Iterative Kalman Filter Track Finding

1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing

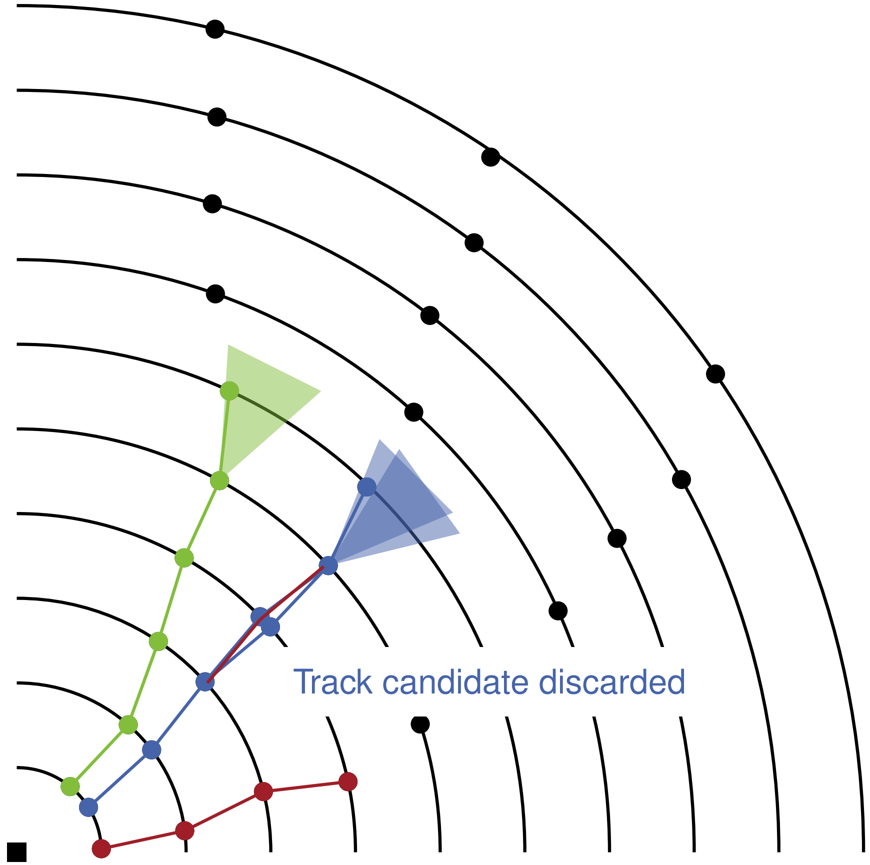


Institute of Theoretical Informatics
Algorithmics Group

Particle Track Reconstruction

Approach most used: Iterative Kalman Filter Track Finding

1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing

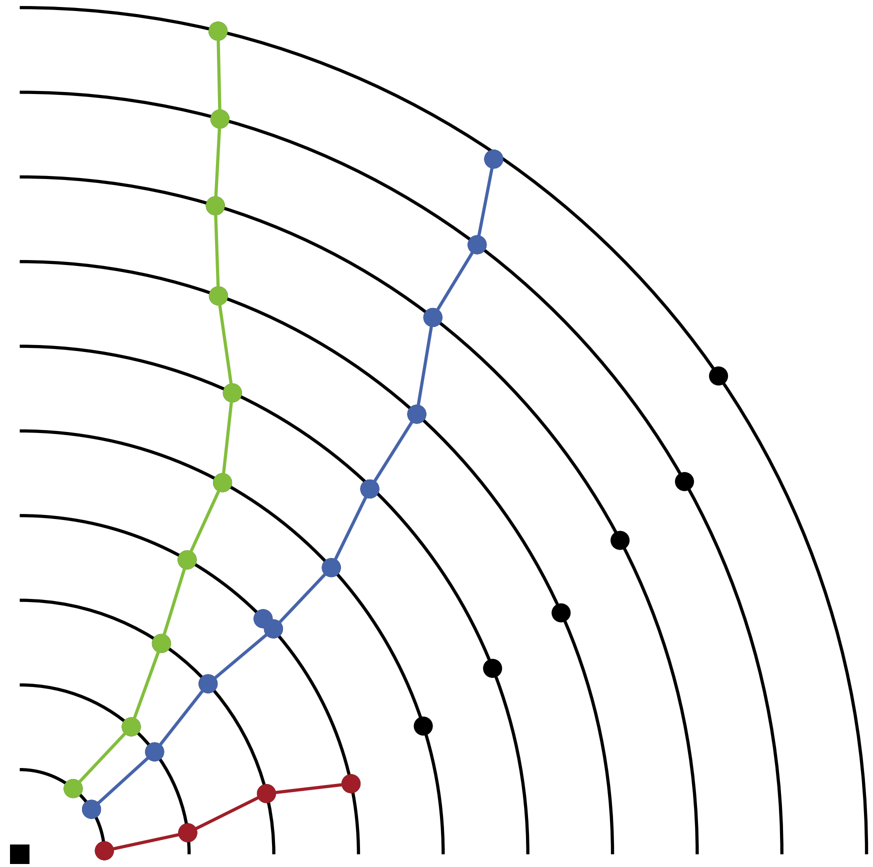


Particle Track Reconstruction

Karlsruhe Institute of Technology

Approach most used: Iterative Kalman Filter Track Finding

1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing

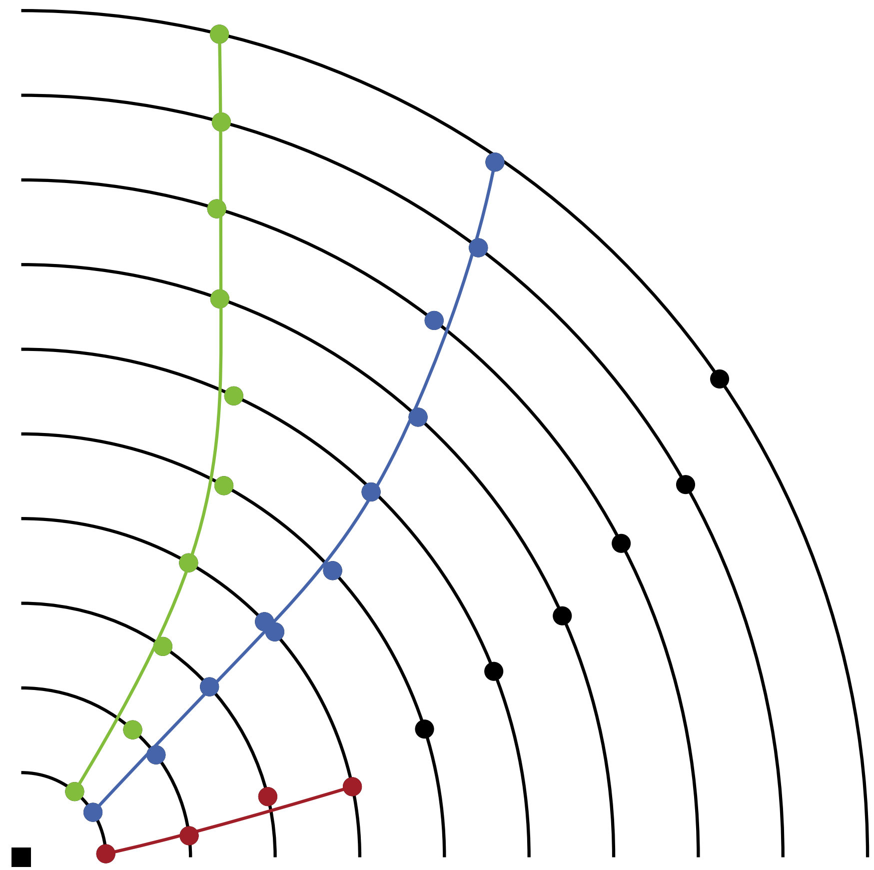


Particle Track Reconstruction

Karlsruhe Institute of Technology

Approach most used: Iterative Kalman Filter Track Finding

1. **Seeding**
   - Find hit triplets in inner layers
   - Rough track parameters
2. **Track Finding**
   - Extrapolate track outwards
   - Extend track by suitable hits
3. **Track Fitting**
   - Estimate track parameter
   - Inward and outward smoothing

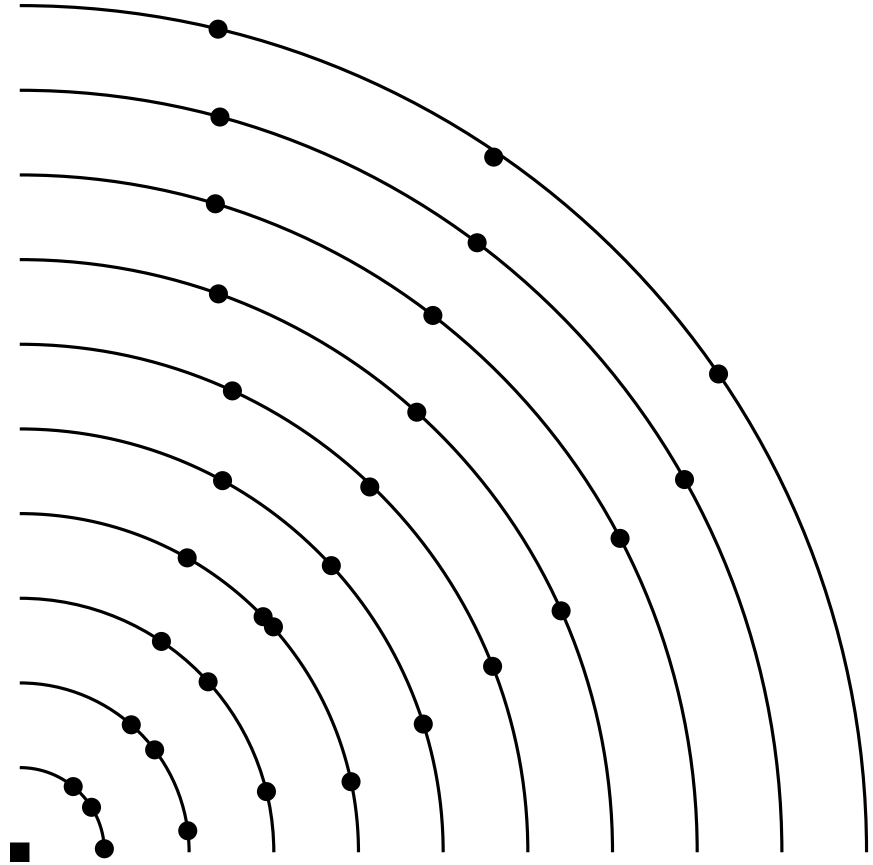


Particle Track Reconstruction

Tracking as graph problem: definition of vertices and edges

$G = (V, E, \omega)$

- Find triplets in all layer combinations
- $V = \{v = (h_1, h_2, h_3)\}$



Institute of Theoretical Informatics
Algorithmics Group

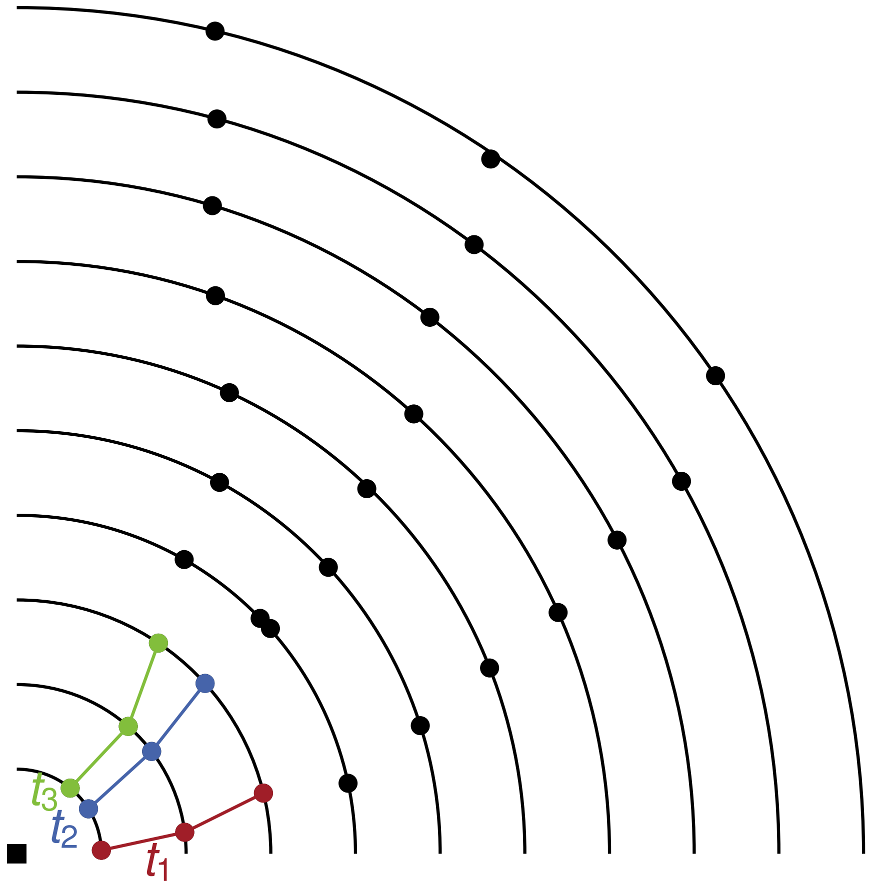
Particle Track Reconstruction

Karlsruhe Institute of Technology

Tracking as graph problem: definition of vertices and edges

$G = (V, E, \omega)$

- Find triplets in all layer combinations
- $V = \{v = (h_1, h_2, h_3)\}$



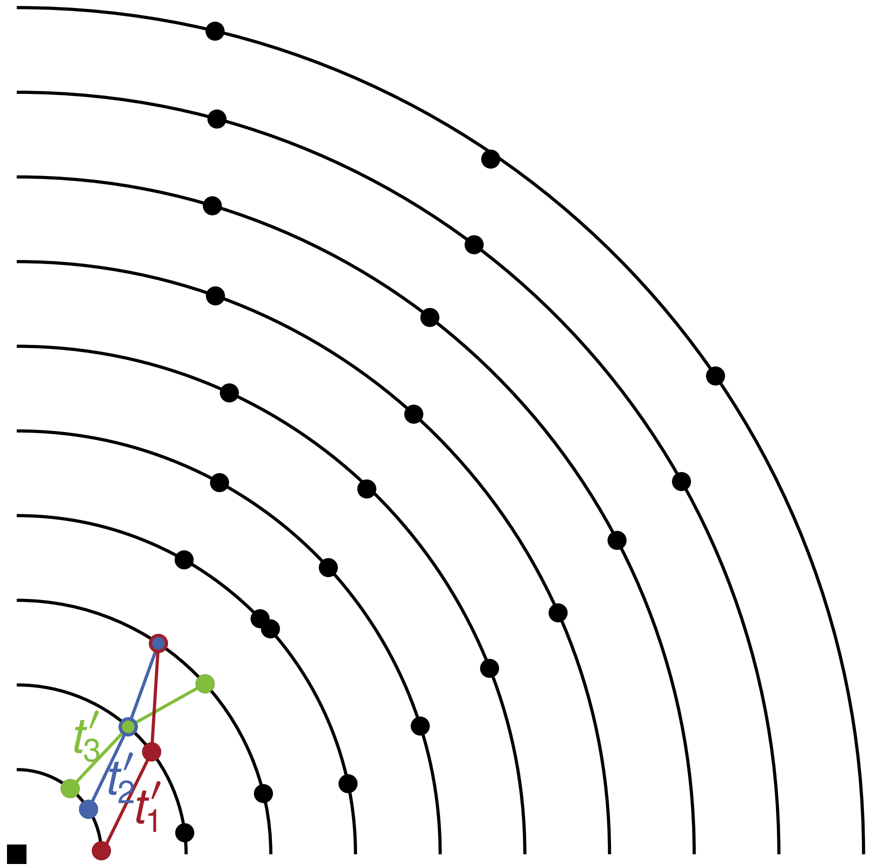
Particle Track Reconstruction

Karlsruhe Institute of Technology

Tracking as graph problem: definition of vertices and edges

$G = (V, E, \omega)$

- Find triplets in all layer combinations
- $V = \{v = (h_1, h_2, h_3)\}$



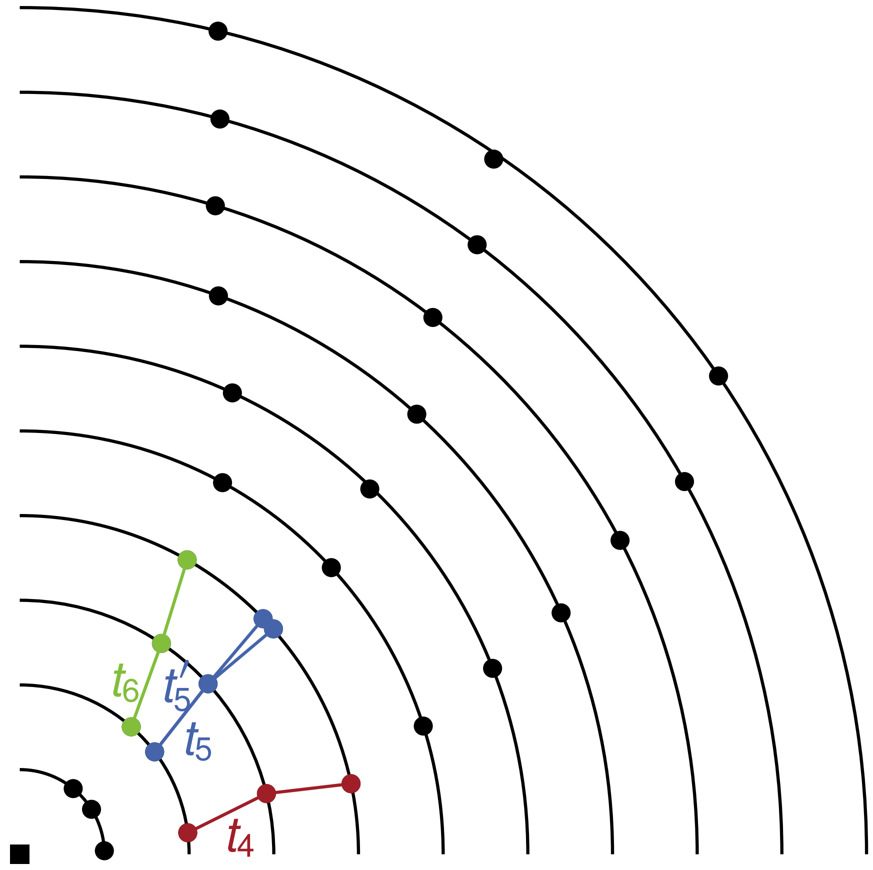
Particle Track Reconstruction

Karlsruhe Institute of Technology

Tracking as graph problem: definition of vertices and edges

$G = (V, E, \omega)$

- Find triplets in all layer combinations
- $V = \{v = (h_1, h_2, h_3)\}$

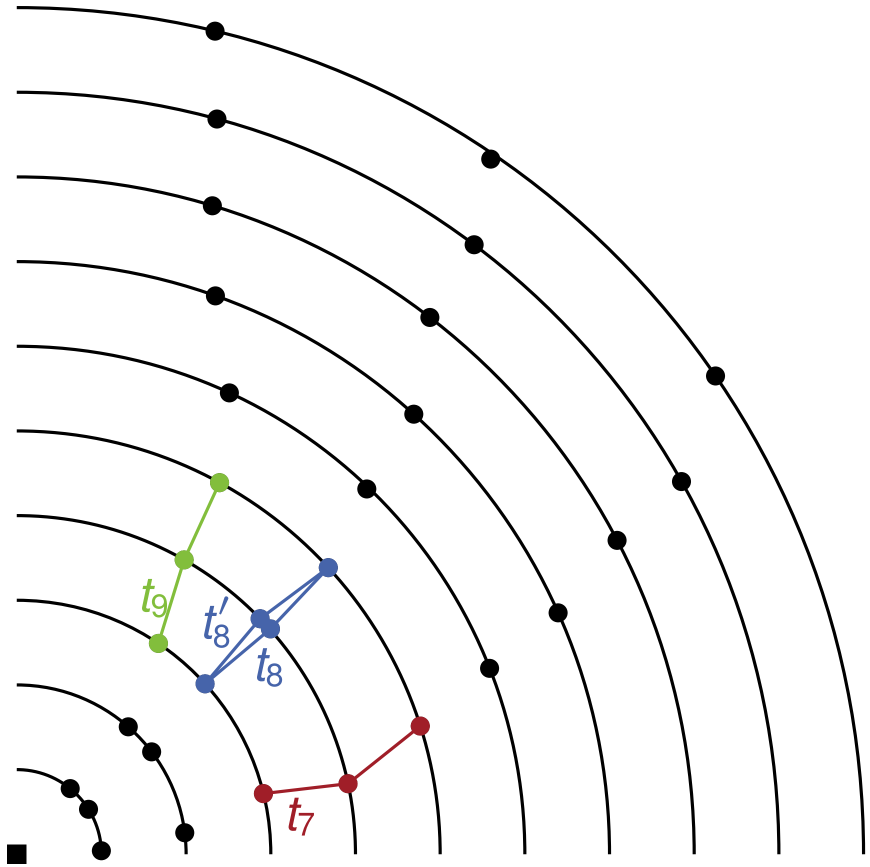


Particle Track Reconstruction

Tracking as graph problem: definition of vertices and edges

$G = (V, E, \omega)$

- Find triplets in all layer combinations
- $V = \{v = (h_1, h_2, h_3)\}$

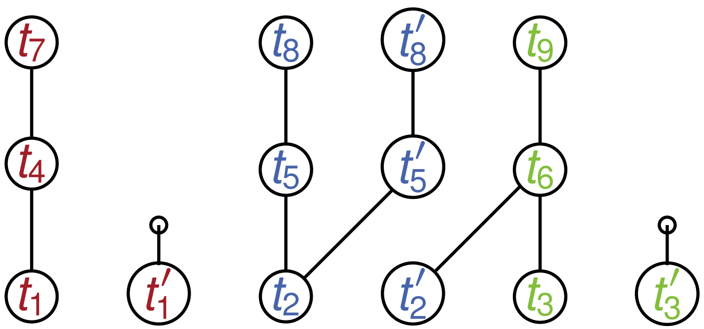


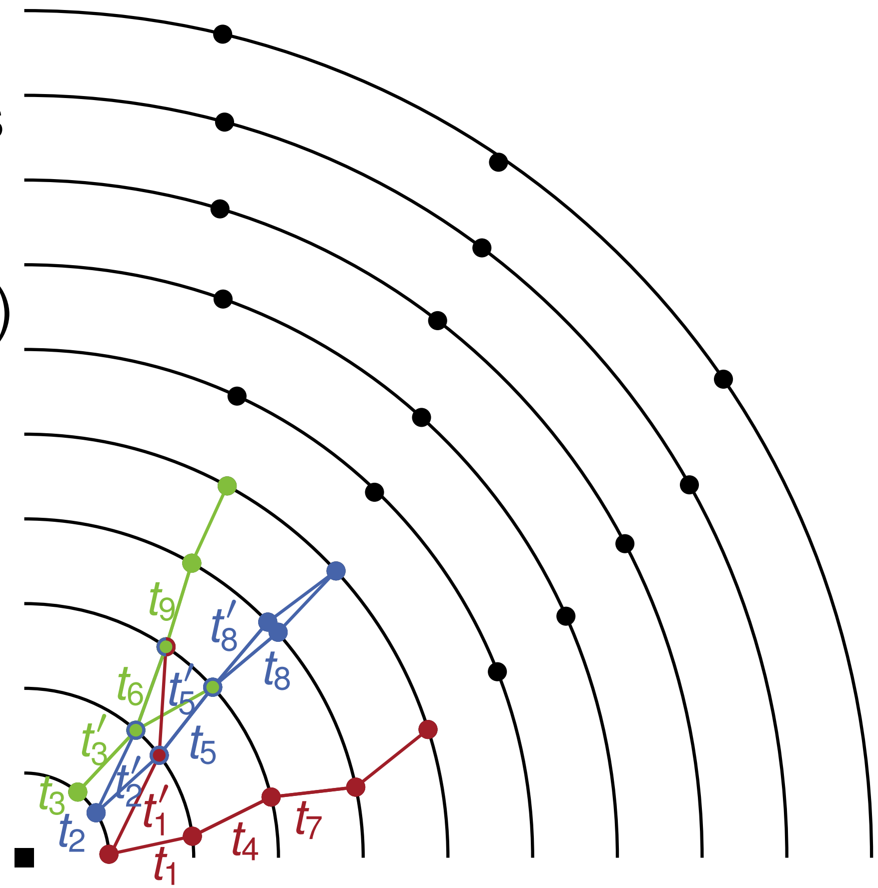
Particle Track Reconstruction

Tracking as graph problem: definition of vertices and edges

$G = (V, E, \omega)$

- Find triplets in all layer combinations
- $V = \{v = (h_1, h_2, h_3)\}$
- Vertices that share one or two hit(s) are connected by edge
- $E = \{e = (v_1, v_2) : v_1 \cap v_2 \neq \emptyset\}$





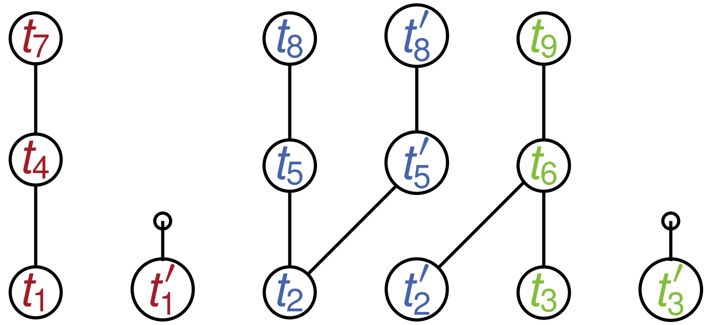
Institute of Theoretical Informatics
Algorithmics Group

Particle Track Reconstruction

Tracking as graph problem: definition of vertices and edges

$G = (V, E, \omega)$

- defining $\omega(e)$ is the hard part, e.g.
  - angular difference $\Delta\phi$, $\Delta\theta$
  - curvature Δc
  - χ^2 of circle fit of all four hits
- solve all-pair-shortest-path problem



Goodish: $O\left(|V|^3\right)$

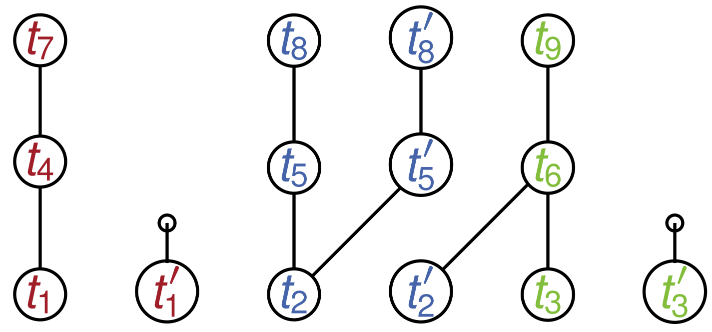
Institute of Theoretical Informatics
Algorithmics Group

Particle Track Reconstruction

Tracking as graph problem: definition of vertices and edges

$G = (V, E, \omega)$

- defining $\omega(e)$ is the hard part, e.g.
  - angular difference $\Delta\phi$, $\Delta\theta$
  - curvature Δc
  - χ^2 of circle fit of all four hits
- solve all-pair-shortest-path problem



Goodish: $O\left(|V|^3\right)$

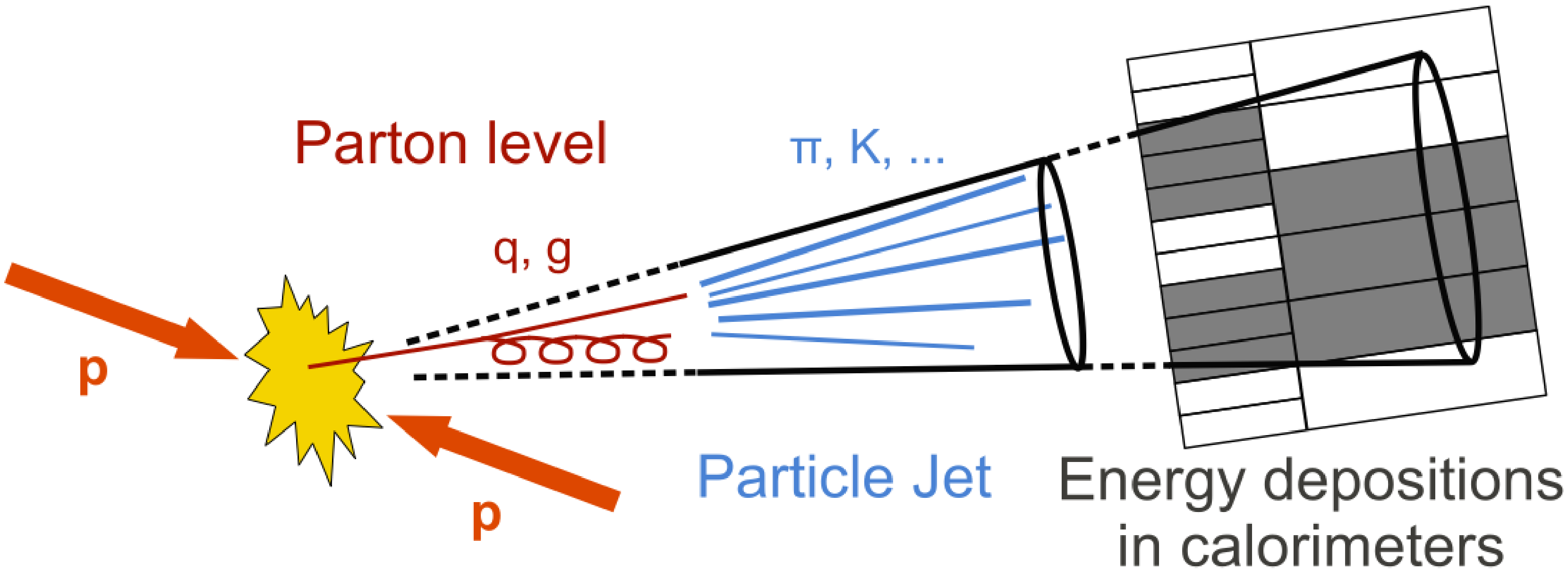
Challenge:
Weight function must ensure that:

- paths corresponding to valid tracks are lighter than others
- otherwise a fake track is reconstructed

Institute of Theoretical Informatics
Algorithmics Group

Jet Clustering

- **Jets:** collimated spray of hadrons from fragmentation of quark or gluon
- reveal direction and energy original "parton"
- jets are reconstructed from particles found in detector
- various algorithms exist to cluster jets from reconstructed particles e.g. k_t algorithm



Jet Clustering

Input: list of particles $\mathbf{P}$

Output: list of jets $\mathbf{J}$

while $\mathbf{P} \neq \emptyset$ **do** ▷ $O(n)$ times

  for $(i, j) \in \mathbf{P} \times \mathbf{P}$ **do** ▷ $O\left(n^2\right)$

    $d_{i,j} = \min(k_{t,i}^2, k_{t,j}^2) \cdot \Delta R_{i,j}^2$

  for $i \in \mathbf{P}$ **do** ▷ $O(n)$

    $d_{i,B} = k_{t,i}^2$

  $d_{\text{min}} = \min(d_{i,j}, d_{i,B})$ ▷ $O\left(n^2\right)$

  if $d_{\text{min}} = d_{i,j}$ **then**

    $i = \text{combine}(i, j), \mathbf{P} \setminus \{j\}$ ▷ merge i and j, delete j

  else

    $\mathbf{J} \cup i, \mathbf{P} \setminus \{i\}$ ▷ finalize jet i

Jet Clustering

Input: list of particles $\mathbf{P}$
Output: list of jets $\mathbf{J}$

```
while P ≠ ∅ do
    for (i, j) ∈ P × P do
        d_{i,j} = min(k_{t,i}^2, k_{t,j}^2) · ΔR_{i,j}^2
    for i ∈ P do
        d_{i,B} = k_{t,i}^2
    d_min = min(d_{i,j}, d_{i,B})
    if d_min = d_{i,j} then
        i = combine(i, j), P \ {j}
    else
        J ∪ i, P \ {i}
```

- $k_{t,i}$: transverse momentum
- $\Delta R_{i,j}^2 = (\eta_i - \eta_j)^2 + (\phi_i - \phi_j)^2$
  - η_i: rapidity
  - ϕ_i: azimuth

Jet Clustering

Input: list of particles $\mathbf{P}$

Output: list of jets $\mathbf{J}$

while $\mathbf{P} \neq \emptyset$ **do** ▷ O(n) times

  for $(i, j) \in \mathbf{P} \times \mathbf{P}$ **do** ▷ $O\left(n^2\right)$

    $d_{i,j} = \min(k_{t,i}^2, k_{t,j}^2) \cdot \Delta R_{i,j}^2$

  for $i \in \mathbf{P}$ **do** ▷ $O(n)$

    $d_{i,B} = k_{t,i}^2$

  $d_{\text{min}} = \min(d_{i,j}, d_{i,B})$ ▷ $O\left(n^2\right)$

Goodish: $O\left(|\mathbf{P}|^3\right)$ prohibitive for high multiplicities

Jet Clustering

Improving the $O(n^3)$ runtime:

Lemma:
If i, j have the smallest $d_{i,j}$ and $k_{t,i} < k_{t,j}$, then $R_{i,j} < R_{i,l}$ for all $l \neq j$.

For minimum $d_{i,j}$: i and j geometrically nearest-neighbors on (η, ϕ)-plane

[Cacciari M. and Salam, G.P., *Dispelling the N^3 myth for the k_t jet-finder*]

Institute of Theoretical Informatics
Algorithmics Group

Jet Clustering

for $i \in \mathbf{P}$ **do**
  $\mathcal{N}_i$ = findNearestNeighbor(i) $\triangleright$ O(n)
  $d_i = \min(k_{t,i}^2, k_{t,\mathcal{N}_i}^2) \cdot \Delta R_{i,\mathcal{N}_i}^2, \; d_{i,B} = k_{t,i}^2$
while $\mathbf{P} \neq \emptyset$ **do** $\triangleright$ O(n) times
  $d_{\min} = \min(d_i, d_{i,B})$ $\triangleright$ O(n)
  if $d_{\min} = d_i$ **then**
    i = combine($i, \mathcal{N}_i$), $\mathbf{P} \setminus \{\mathcal{N}_i\}$ $\triangleright$ merge i and $\mathcal{N}_i$, delete $\mathcal{N}_i$
  else
    $\mathbf{J} \cup i, \mathbf{P} \setminus \{i\}$ $\triangleright$ finalize jet i
  for particles j with $\mathcal{N}_j = i$ **do** $\triangleright$ O(1) many
    $\mathcal{N}_j$ = findNearestNeighbor(j)
  for $j \in \mathbf{P}$ **do**
    $\mathcal{N}_j$ = updateNearestNeighbor(j, i) $\triangleright$ O(1)

Jet Clustering

for $i \in \mathbf{P}$ **do**
    $\mathcal{N}_i$ = findNearestNeighbor(i) ▷ O(n)
    $d_i = \min(k_{t,i}^2, k_{t,\mathcal{N}_i}^2) \cdot \Delta R_{i,\mathcal{N}_i}^2,\ d_{i,B} = k_{t,i}^2$
while $\mathbf{P} \neq \emptyset$ **do** ▷ O(n) times
    $d_{\min} = \min(d_i, d_{i,B})$ ▷ O(n)
    if $d_{\min} = d_i$ **then**
        $i = \text{combine}(i, \mathcal{N}_i), \mathbf{P} \setminus \{\mathcal{N}_i\}$ ▷ merge i and $\mathcal{N}_i$, delete $\mathcal{N}_i$
    else

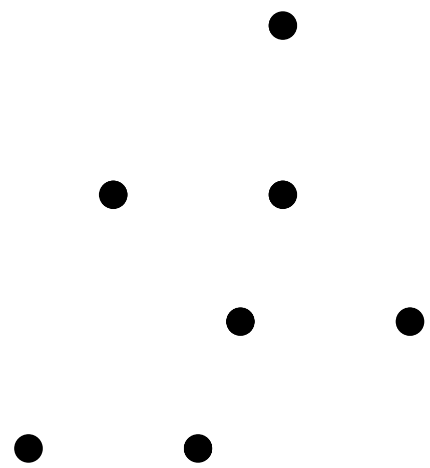
Goodish: O$\left(|\mathbf{P}|^2\right)$ – but we can do better!

            $\mathcal{N}_j$ = updateNearestNeighbor(j, i) ▷ O(1)

Institute of Theoretical Informatics
Algorithmics Group

Jet Clustering

Enter geometric graphs. Given a point set $\mathbf{P}$ in $\mathbb{R}^2$

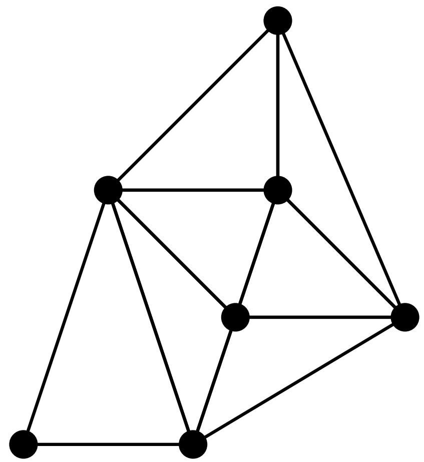


Jet Clustering

Enter geometric graphs. Given a point set $\mathbf{P}$ in $\mathbb{R}^2$

A triangulation $T(\mathbf{P})$ is the subdivision of the convex hull of $\mathbf{P}$ into triangles such that

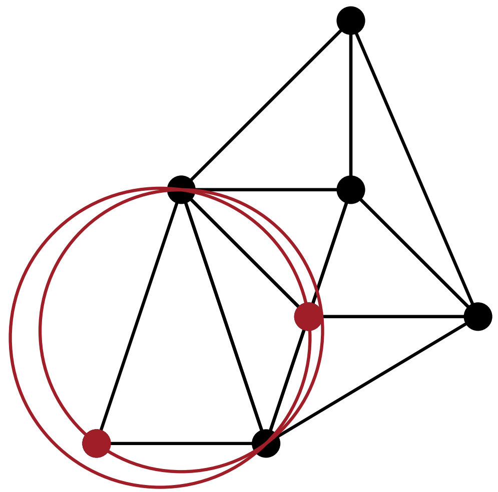
- the vertices of $T(\mathbf{P})$ coincide with $\mathbf{P}$
- any two triangles of $T(\mathbf{P})$ intersect in a common edge or not at all



Jet Clustering

Enter geometric graphs. Given a point set $\mathbf{P}$ in $\mathbb{R}^2$

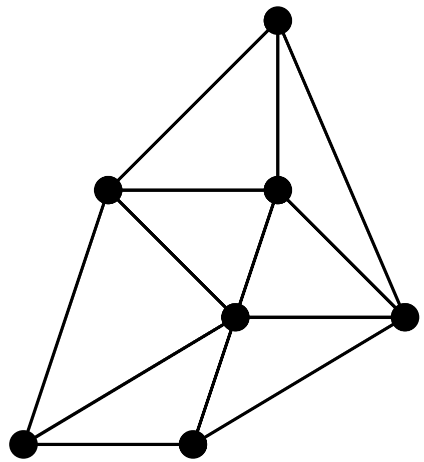
A Delaunay triangulation $DT(\mathbf{P})$ is a triangulation such that no point of $\mathbf{P}$ is inside the circumcircle of any simplex of $DT(\mathbf{P})$.



Jet Clustering

Enter geometric graphs. Given a point set $\mathbf{P}$ in $\mathbb{R}^2$

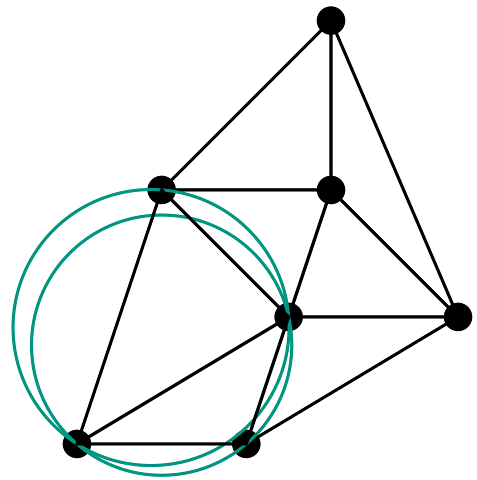
A Delaunay triangulation $DT(\mathbf{P})$ is a triangulation such that no point of $\mathbf{P}$ is inside the circumcircle of any simplex of $DT(\mathbf{P})$.



Jet Clustering

Enter geometric graphs. Given a point set $\mathbf{P}$ in $\mathbb{R}^2$

A Delaunay triangulation $DT(\mathbf{P})$ is a triangulation such that no point of $\mathbf{P}$ is inside the circumcircle of any simplex of $DT(\mathbf{P})$.



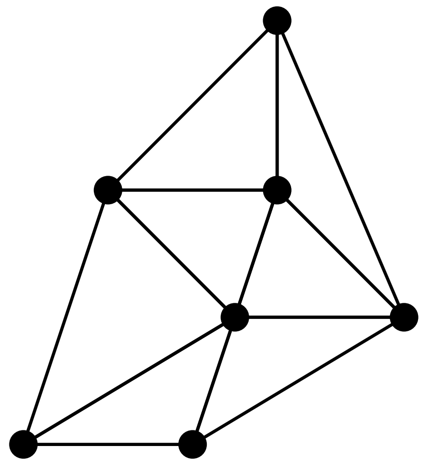
Institute of Theoretical Informatics
Algorithmics Group

Jet Clustering

Enter geometric graphs. Given a point set $\mathbf{P}$ in $\mathbb{R}^2$

A Delaunay triangulation $DT(\mathbf{P})$ is a triangulation such that no point of $\mathbf{P}$ is inside the circumcircle of any simplex of $DT(\mathbf{P})$.

- nearest-neighbor graph of $\mathbf{P}$ is a subgraph of $DT(\mathbf{P})$
- $DT(\mathbf{P})$ can be constructed in $O(n \log n)$
- $DT(\mathbf{P})$ can be updated in $O(\log n)$



Institute of Theoretical Informatics
Algorithmics Group

Jet Clustering

construct $DT(\mathbf{P})$ $\triangleright$ $O(n \log n)$
for $i \in \mathbf{P}$ **do**
    $d_i = \min(k_{t,i}^2, k_{t,\mathcal{N}_i}^2) \cdot \Delta R_{i,\mathcal{N}_i}^2,\ d_{i,B} = k_{t,i}^2$ $\triangleright$ O(1)
construct binary binary trees T_{d_i}, $T_{d_{i,B}}$ $\triangleright$ $O(n \log n)$
while $\mathbf{P} \neq \emptyset$ **do** $\triangleright$ O(n) times
    $d_{\min} = \min(d_i, d_{i,B})$ $\triangleright$ $O(\log n)$
    if $d_{\min} = d_i$ **then**
        $i = \text{combine}(i, \mathcal{N}_i), \mathbf{P} \setminus \{\mathcal{N}_i\}$ $\triangleright$ merge i and $\mathcal{N}_i$, delete $\mathcal{N}_i$
    else
        $\mathbf{J} \cup i, \mathbf{P} \setminus \{i\}$ $\triangleright$ finalize jet i
    update $DT(\mathbf{P})$ $\triangleright$ $O(\log n)$
    update T_{d_i}, $T_{d_{i,B}}$ $\triangleright$ $O(\log n)$

Jet Clustering

construct $DT(\mathbf{P})$ ▷ $O(n \log n)$
for $i \in \mathbf{P}$ **do**
    $d_i = \min(k_{t,i}^2, k_{t,\mathcal{N}_i}^2) \cdot \Delta R_{i,\mathcal{N}_i}^2, \; d_{i,B} = k_{t,i}^2$ ▷ O(1)
construct binary binary trees $T_{d_i}, T_{d_{i,B}}$ ▷ $O(n \log n)$
while $\mathbf{P} \neq \emptyset$ **do** ▷ O(n) times
    $d_{\min} = \min(d_i, d_{i,B})$ ▷ $O(\log n)$
    if $d_{\min} = d_i$ **then**
        $i = \text{combine}(i, \mathcal{N}_i), \mathbf{P} \setminus \{\mathcal{N}_i\}$ ▷ merge i and $\mathcal{N}_i$, delete $\mathcal{N}_i$

Good: $\mathrm{O}\big(|\mathbf{P}| \log |\mathbf{P}|\big)$

Tutorial

Institute of Theoretical Informatics
Algorithmics Group

Credits

The slides of this course are partially based on the following lectures/talks:

- P. Sanders - Algorithmen I
- P. Sanders - Algorithmen II
- P. Sanders, R. van Stee - Approximations- und Online-Algorithmen
- C. Schulz - Graphpartitionierung und Graphenclustern in Theorie und Praxis
- H. Meyerhenke - Algorithmische Methoden zur Netzwerkanalyse
- Henning Meyerhenke - NetworKit: A Parallel Interactive Tool Suite for Analyzing Massive Networks
- H. Meyerhenke - Network Analysis with NetworKit: Interactive, Feature-rich, Fast
- S. Schlag - k-way Hypergraph Partitioning via n-Level Recursive Bisection
- D. Funke - Parallel Triplet Finding for Particle Track Reconstruction