Introduction to Optimization
Greedy Algorithms

October 28, 2016
École Centrale Paris, Châtenay-Malabry, France

INVENTORS FOR THE DIGITAL WORLD

Dimo Brockhoff
Inria Saclay – Ile-de-France

Course Overview

Date		Topic
Fri, 7.10.2016		Introduction
Fri, 28.10.2016	D	Introduction to Discrete Optimization + Greedy algorithms I
Fri, 4.11.2016	D	Greedy algorithms II + Branch and bound
Fri, 18.11.2016	D	Dynamic programming
Mon, 21.11.2016 in S103-S105	D	Approximation algorithms and heuristics
Fri, 25.11.2016 in S103-S105	C	Introduction to Continuous Optimization I
Mon, 28.11.2016	C	Introduction to Continuous Optimization II
Mon, 5.12.2016	C	Gradient-based Algorithms
Fri, 9.12.2016	C	Stochastic Optimization and Derivative Free Optimization I
Mon, 12.12.2016	C	Stochastic Optimization and Derivative Free Optimization II
Fri, 16.12.2016	C	Benchmarking Optimizers with the COCO platform
Wed, 4.1.2017		Exam

all classes last 3h15 and take place in S115-S117 (see exceptions)

Introduction to Discrete Optimization

Discrete Optimization

Discrete optimization:

- discrete variables
- or optimization over discrete structures (e.g. graphs)
- search space often finite, but typically too large for enumeration
- → need for smart algorithms

Algorithms for discrete problems:

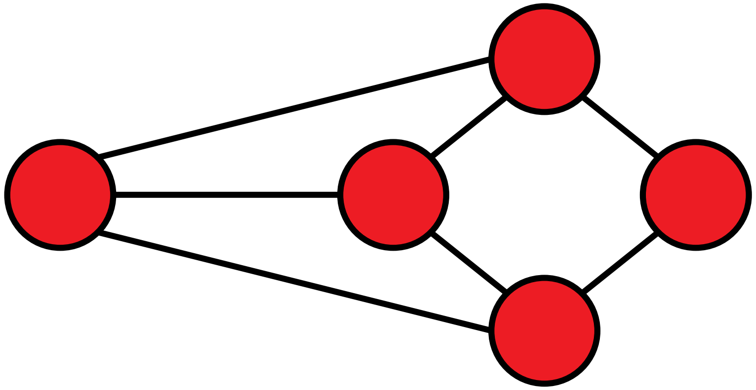
- typically problem-specific
- but some general concepts are repeatedly used:
  - greedy algorithms (lecture 2 today)
  - branch&bound (lecture 3)
  - dynamic programming (lecture 4)
  - heuristics (lecture 5)

Basic Concepts of Graph Theory

[following for example http://math.tut.fi/~ruohonen/GT_English.pdf]

Graphs

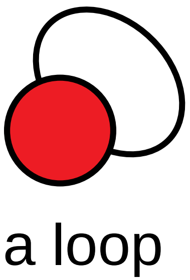
Definition 1 *An undirected graph G is a tupel $G = (V, E)$ of edges $e = \{u, v\} \in E$ over the vertex set V (i.e., $u, v \in V$).*

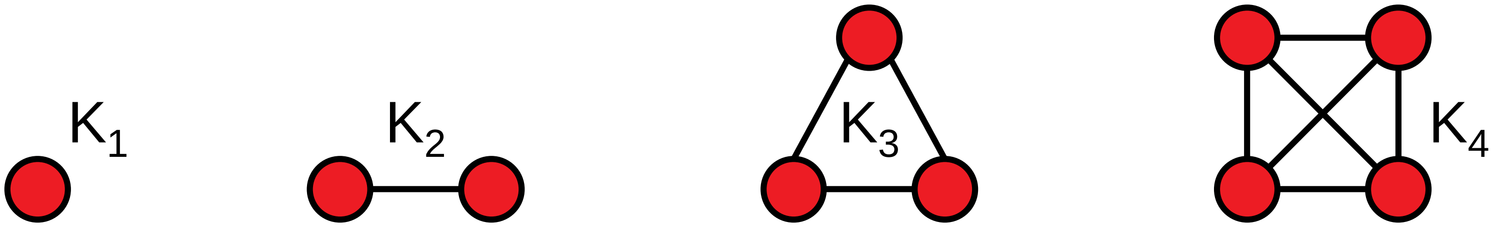


- vertices = nodes
- edges = lines
- Note: edges cover two *unordered* vertices (*undirected* graph)
  - if they are *ordered*, we call G a *directed* graph

Graphs: Basic Definitions

- G is called *empty* if E empty
- u and v are *end vertices* of an edge {u,v}
- Edges are *adjacent* if they share an end vertex
- Vertices u and v are *adjacent* if {u,v} is in E
- The *degree* of a vertex is the number of times it is an end vertex
- A complete graph contains all possible edges (once):



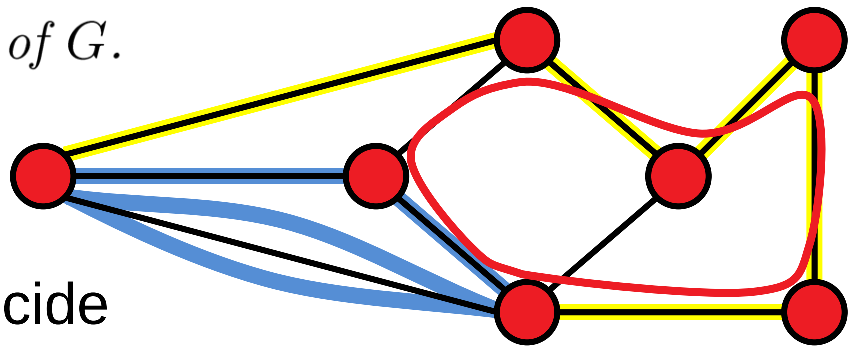


Walks, Paths, and Circuits

Definition 1 *A* walk *in a graph* $G = (V, E)$ *is a sequence*

$$v_{i_0}, e_{i_1} = (v_{i_0}, v_{i_1}), v_{i_1}, e_{i_2} = (v_{i_1}, v_{i_2}), \ldots, e_{i_k}, v_{i_k},$$

alternating vertices and adjacent edges of G.



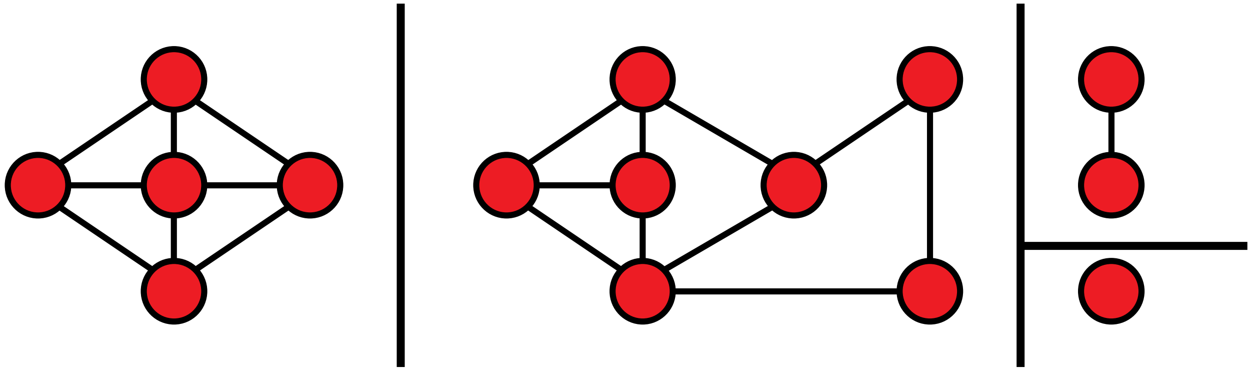
A walk is

- *closed* if first and last node coincide
- a *trail* if each edge traversed at most once
- a *path* if each vertex is visited at most once

- a closed path is a *circuit* or *cycle*
- a closed path involving all vertices of G is a *Hamiltonian cycle*

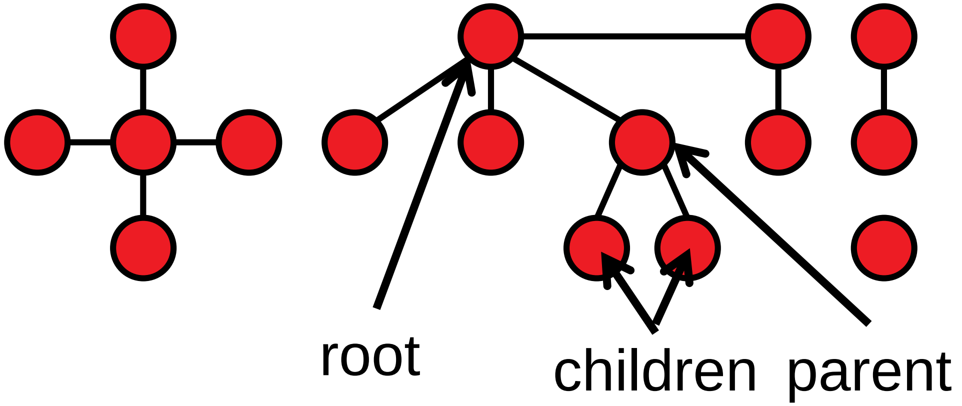
- Two vertices are called *connected* if there is a walk between them in G
- If all vertex pairs in G are connected, G is called connected

- The *connected components* of G are the (maximal) subgraphs which are connected.

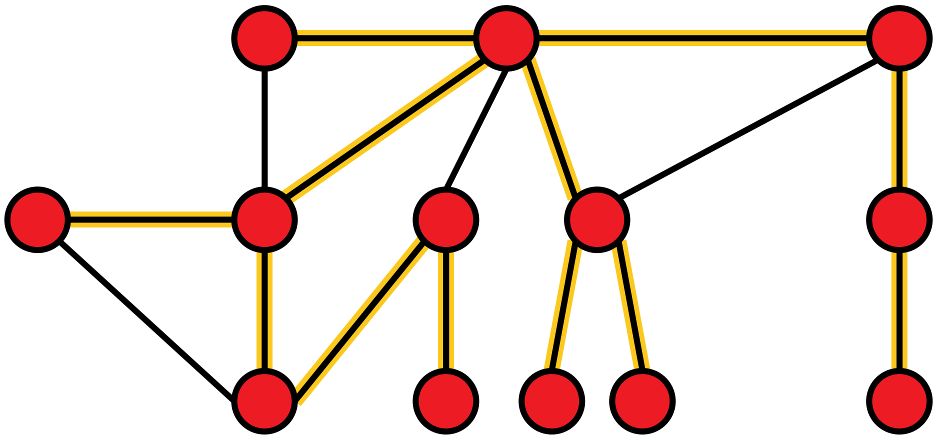


Trees and Forests

- A *forest* is a cycle-free graph
- A *tree* is a connected forest



A *spanning tree* of a connected graph G is a tree in G which contains all vertices of G



Depth-First Search (DFS)

Sometimes, we need to traverse a graph, e.g. to find certain vertices

Depth-first search and breadth-first search are two algorithms to do so

Depth-first Search (for undirected/acyclic and connected graphs)

❶ start at any node x; set i=0

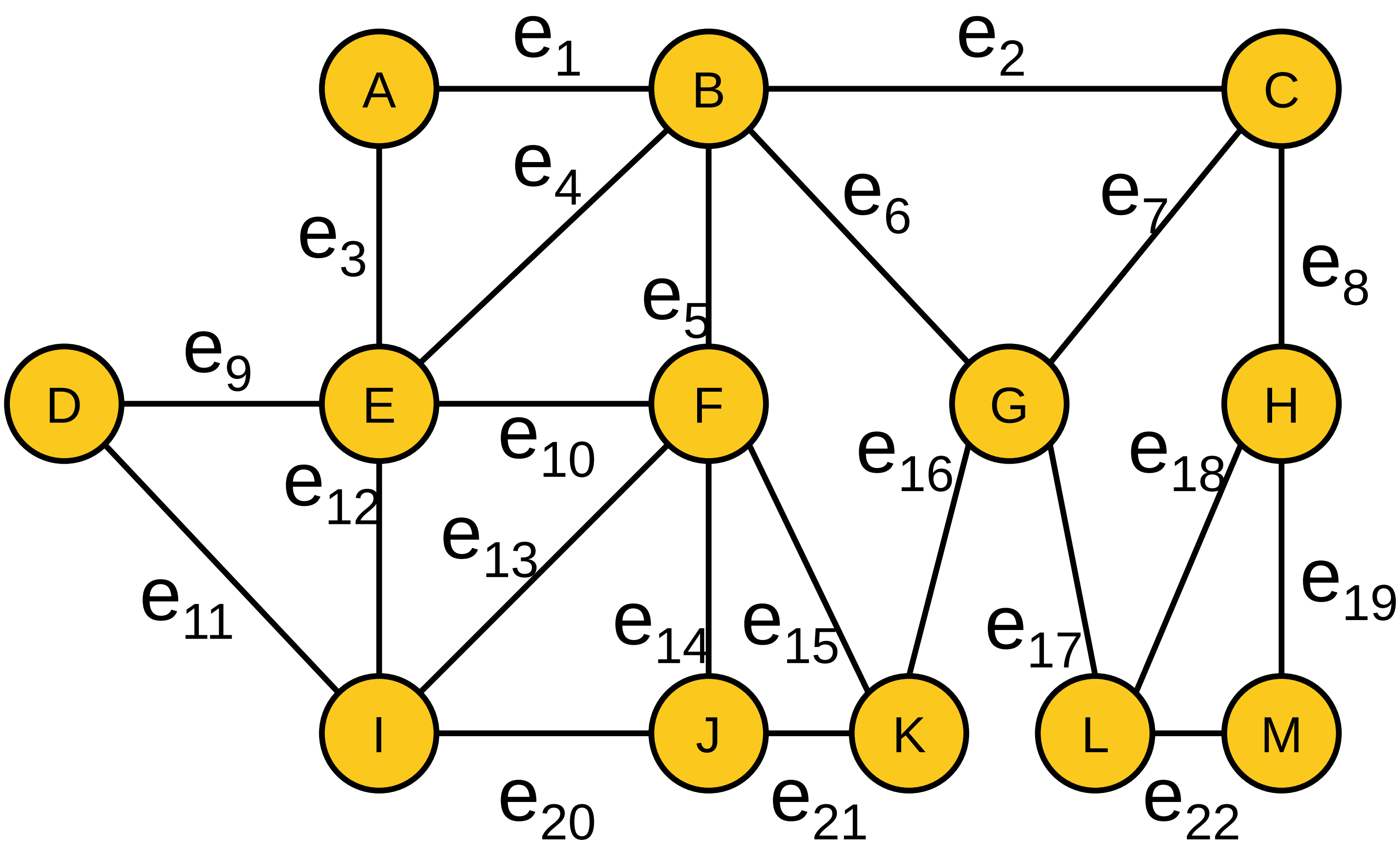
❷ as long as there are unvisited edges {x,y}:

- choose the next unvisited edge {x,y} to a vertex y and mark x as the parent of y
- if y has not been visited so far: i=i+1, give y the number i, and continue the search at x=y in step 2
- else continue with next unvisited edge of x

❸ if all edges {x,y} are visited, we continue with x=parent(x) at step 2 or stop if x==v0

DFS: Stage Exercise

Exercise the DFS algorithm on the following graph!



Breadth-First Search (BFS)

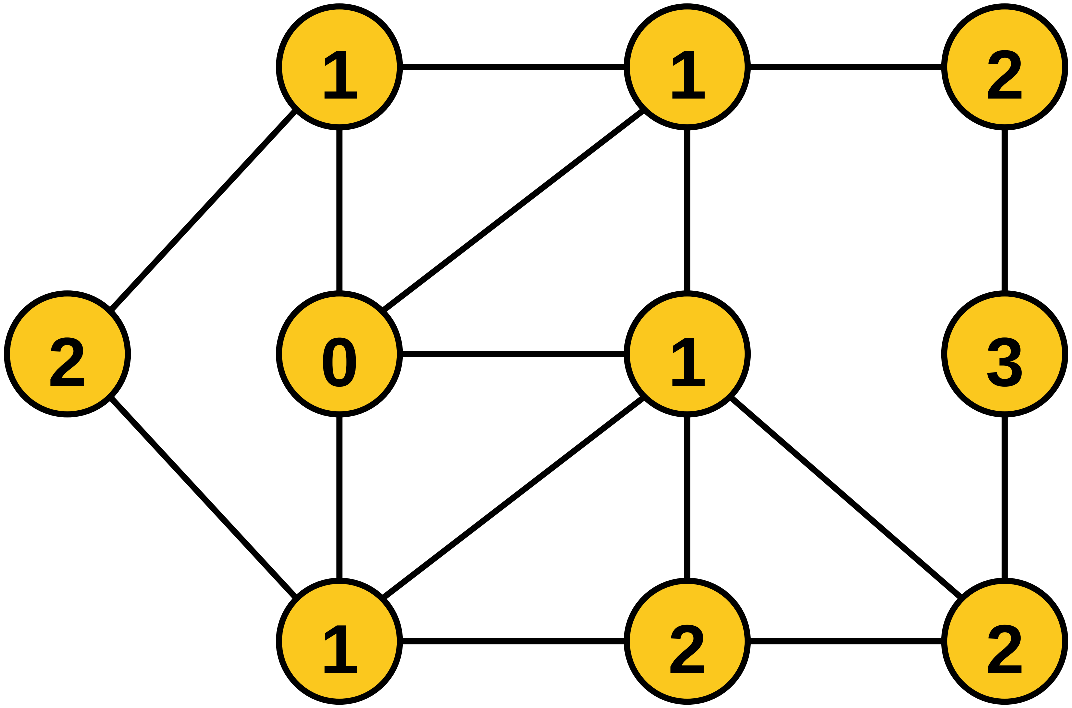
Breadth-first Search (for undirected/acyclic and connected graphs)

❶ start at any node x, set i=0, and label x with value i

❷ as long as there are unvisited edges {x,y} which are adjacent to a vertex x that is labeled with value i:

- label all vertices y with value i+1

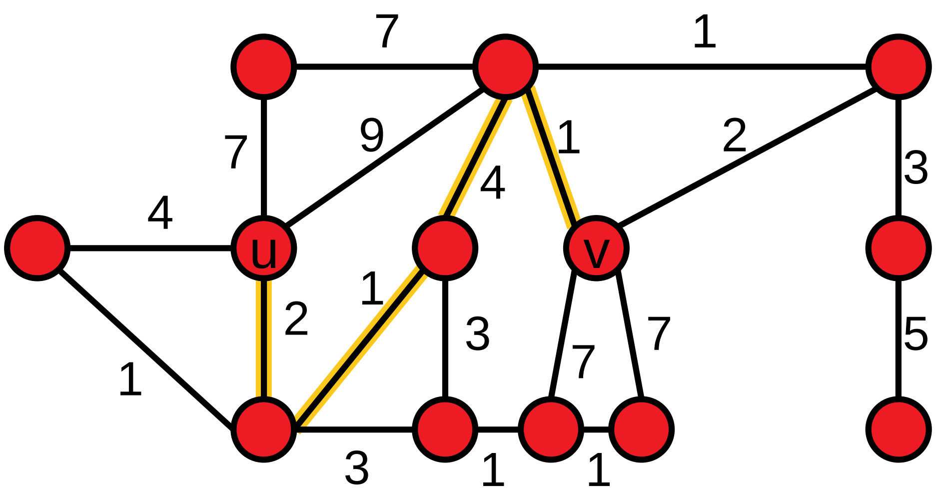
❸ set i=i+1 and go to step 2



Definition of Some Combinatorial Problems Used Later on in the Lecture

Shortest Path problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i.
Find the shortest path from a vertex v to a vertex u, i.e., the path (v, e_1={v, v_1}, v_1, ..., v_k, e_k={v_k,u}, u) such that $w_1 + ... + w_k$ is minimized.



Obvious Applications

Google maps

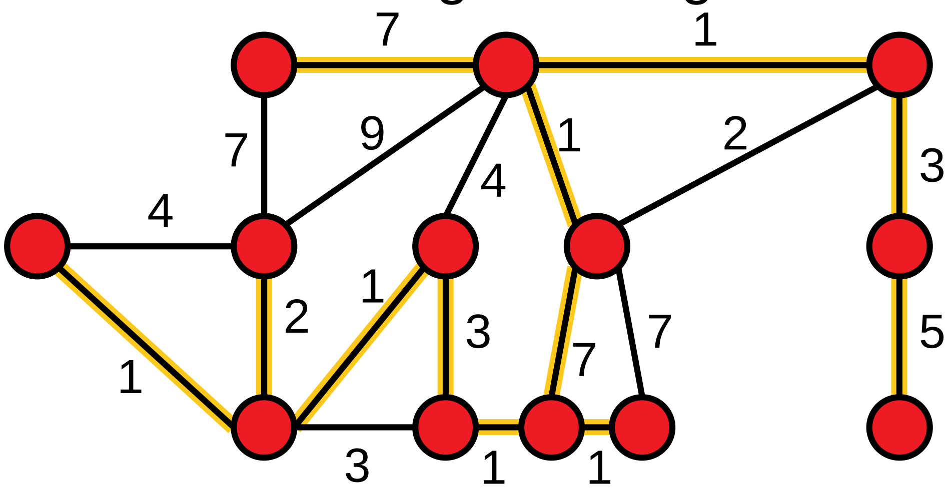
Finding routes for packages in a computer network

...

Minimum Spanning Trees (MST)

Minimum Spanning Tree problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i.
Find the spanning tree with the smallest weight among all spanning trees.



Applications

Setting up a new wired telecommunication/water supply/electricity network

Constructing minimal delay trees for broadcasting in networks

Set Cover Problem (SCP)

Set Cover Problem

Given a set U={1, 2, 3, ..., n}, called the universe, and a set $S=\{s_1, ..., s_m\}$ of m subsets of U, the union of which equals U. Find the smallest subset of S, the union of which also equals U. In other words, find an index $I \subseteq \{1, ..., m\}$ which minimizes $\sum_{i \in I} |s_i|$ such that the union of the s_i ($i \in I$) equals U.

U = {1,2,3,4,5}
S = {{1,2}, {1,3,5}, {1,2,3,5}, {2,3,4}}

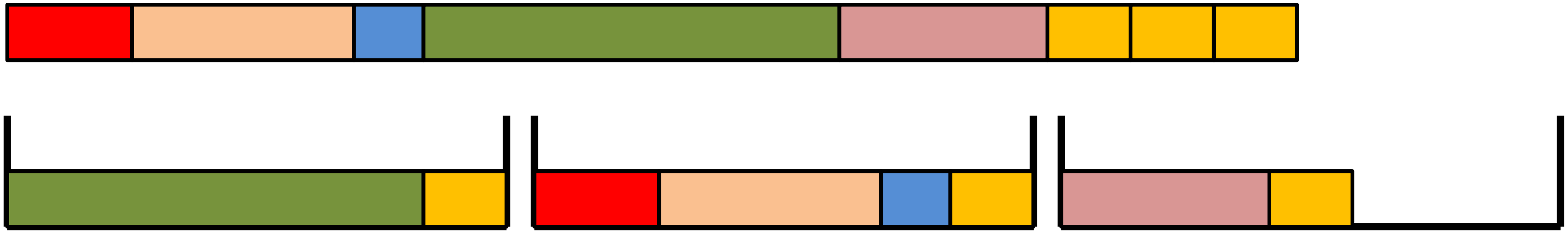
minimal set cover: {1,3,5} {2,3,4}

Application example

IBM's Antivirus use(d) set cover to search for a minimal set of code snippets which appear in all known viruses but not in "good" code

Bin Packing Problem

Given a set of n items with sizes a_1, a_2, ..., a_n. Find an assignment of the a_i's to bins of size V such that the number of bins is minimal and the sum of the sizes of all items assigned to each bin is $\leq$ V.



Applications

similar to multiprocessor scheduling of n jobs to m processors

Integer Linear Programming (ILP)

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \\ \text{and} \quad & x \in \mathbb{Z}^n \end{aligned}$$

- rather a problem class
- can be written as ILP: SAT, TSP, Vertex Cover, Set Packing, ...

Conclusions I

- many, many more problems out there
- typically in practice: need to solve very specific instances
- here only possible to provide you
  - the basic algorithm design ideas
  - applied to a few standard problem classes
  - regular training (i.e. exercises) to gain intuition and experience
  - a broad overview on optimization topics to potentially draw your interest (e.g. towards a PhD on that topic)

Conclusions II

I hope it became clear so far...

...what optimization is about

...what is a graph, a node/vertex, an edge, ...

...and that designing a good algorithm is an important task

Greedy Algorithms

Greedy Algorithms

From Wikipedia:

> "A *greedy algorithm* is an algorithm that follows the problem solving *heuristic* of making the locally optimal choice at each stage with the hope of finding a global optimum."

- Note: typically greedy algorithms do not find the global optimum
- We will see later when this is the case

Greedy Algorithms: Lecture Overview

- Example 1: Money Change
- Example 2: Packing Circles in Triangles
- Example 3: Minimal Spanning Trees (MST) and the algorithm of Kruskal
- The theory behind greedy algorithms: a brief introduction to matroids
- Exercise: A Greedy Algorithm for the Knapsack Problem

Example 1: Money Change

Change-making problem

- Given n coins of distinct values $w_1=1, w_2, ..., w_n$ and a total change W (where $w_1, ..., w_n$, and W are integers).
- Minimize the total amount of coins Σx_i such that $\Sigma w_i x_i = W$ and where x_i is the number of times, coin i is given back as change.

Greedy Algorithm

Unless total change not reached:

add the largest coin which is not larger than the remaining amount to the change

Note: only optimal for standard coin sets, not for arbitrary ones!

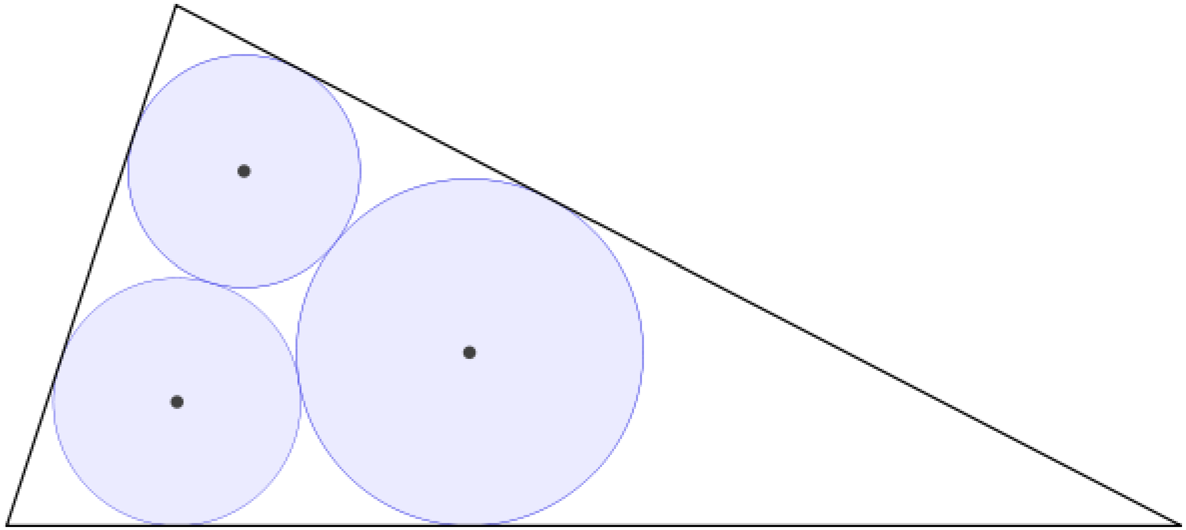
Related Problem:

finishing darts (from 501 to 0 with 9 darts)

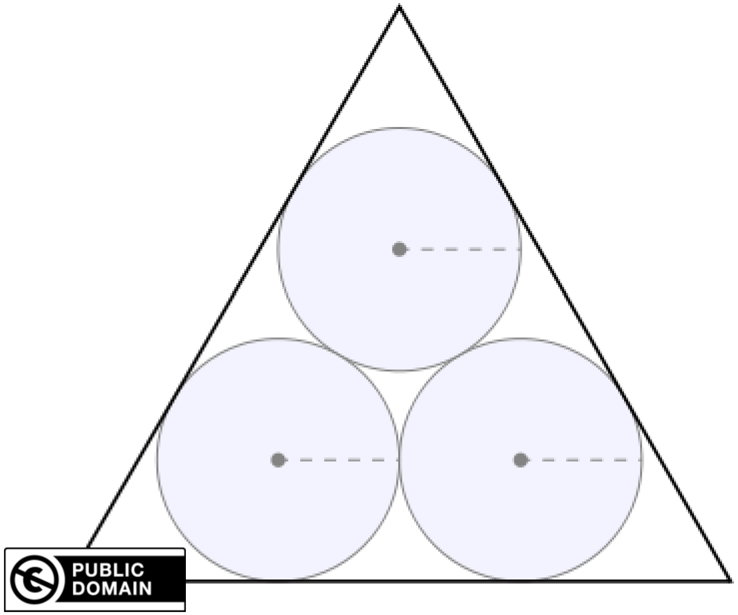
Example 2: Packing Circles in Triangles

G. F. Malfatti posed the following problem in 1803:

- how to cut three cylindrical columns out of a triangular prism of marble such that their total volume is maximized?
- his best solutions were so-called Malfatti circles in the triangular cross-section:
  - all circles are tangent to each other
  - two of them are tangent to each side of the triangle

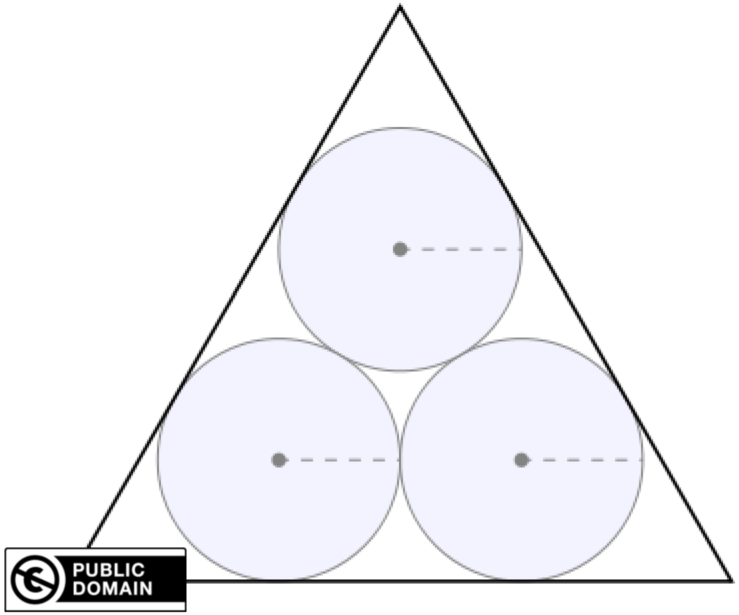


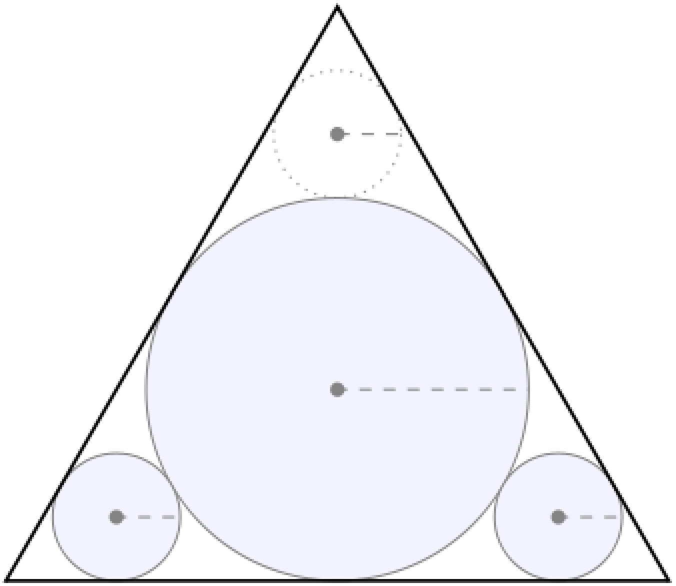
Example 2: Packing Circles in Triangles



What would a greedy algorithm do?

Example 2: Packing Circles in Triangles





What would a greedy algorithm do?

Note that Zalgaller and Los' showed in 1994 that the greedy algorithm is optimal [1]

[1] Zalgaller, V.A.; Los', G.A. (1994), "The solution of Malfatti's problem", *Journal of Mathematical Sciences* **72** (4): 3163–3177, doi:10.1007/BF01249514.

Example 3: Minimal Spanning Trees (MST)

Outline:

- reminder of problem definition
- Kruskal's algorithm
  - including correctness proofs and analysis of running time

MST: Reminder of Problem Definition

A *spanning tree* of a connected graph G is a tree in G which contains all vertices of G

Minimum Spanning Tree Problem (MST):

Given a (connected) graph G=(V,E) with edge weights w_i for each edge e_i. Find a spanning tree T that minimizes the weights of the contained edges, i.e. where

$$\sum_{e_i \in T} w_i$$

is minimized.

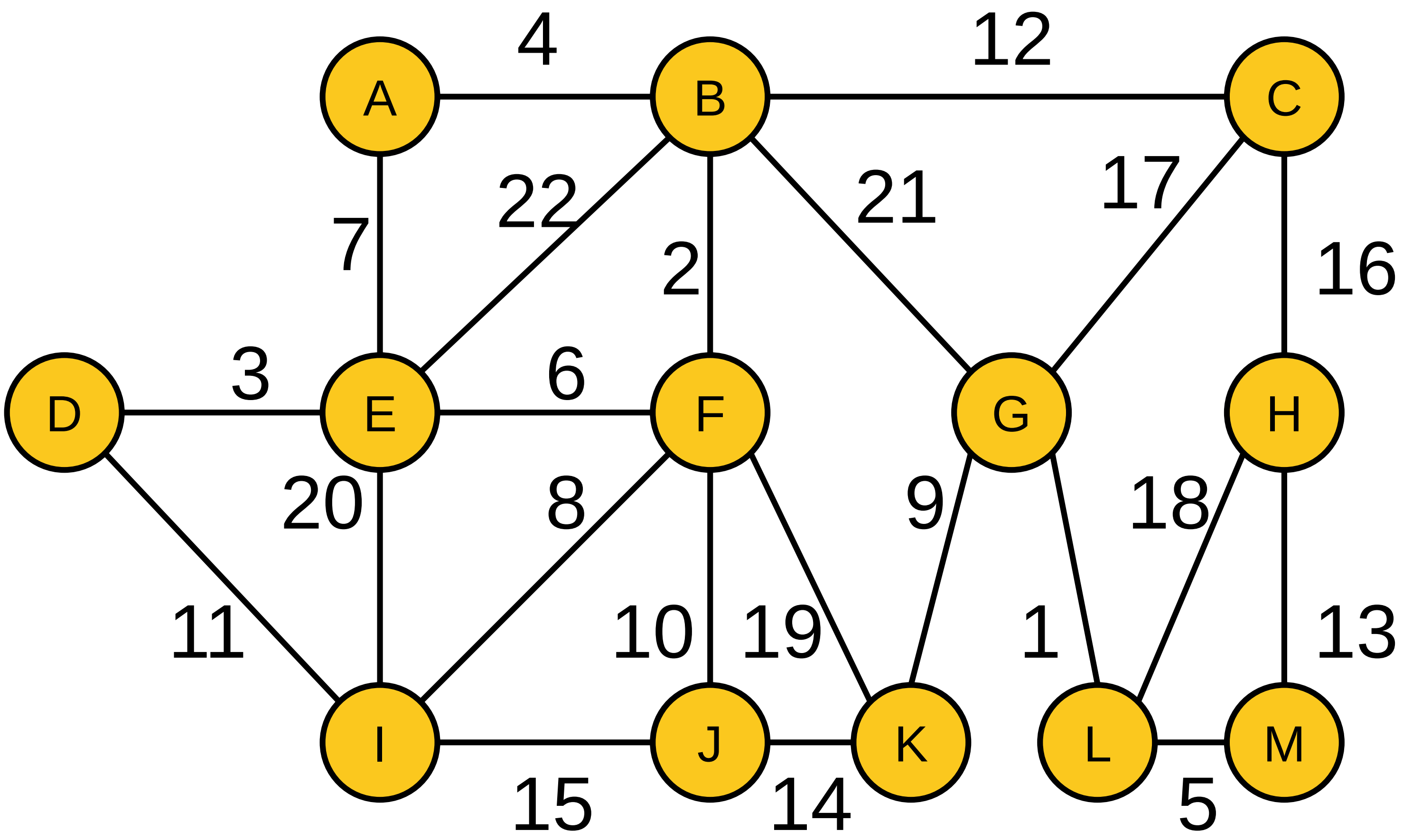
Kruskal's Algorithm

Algorithm, see [1]

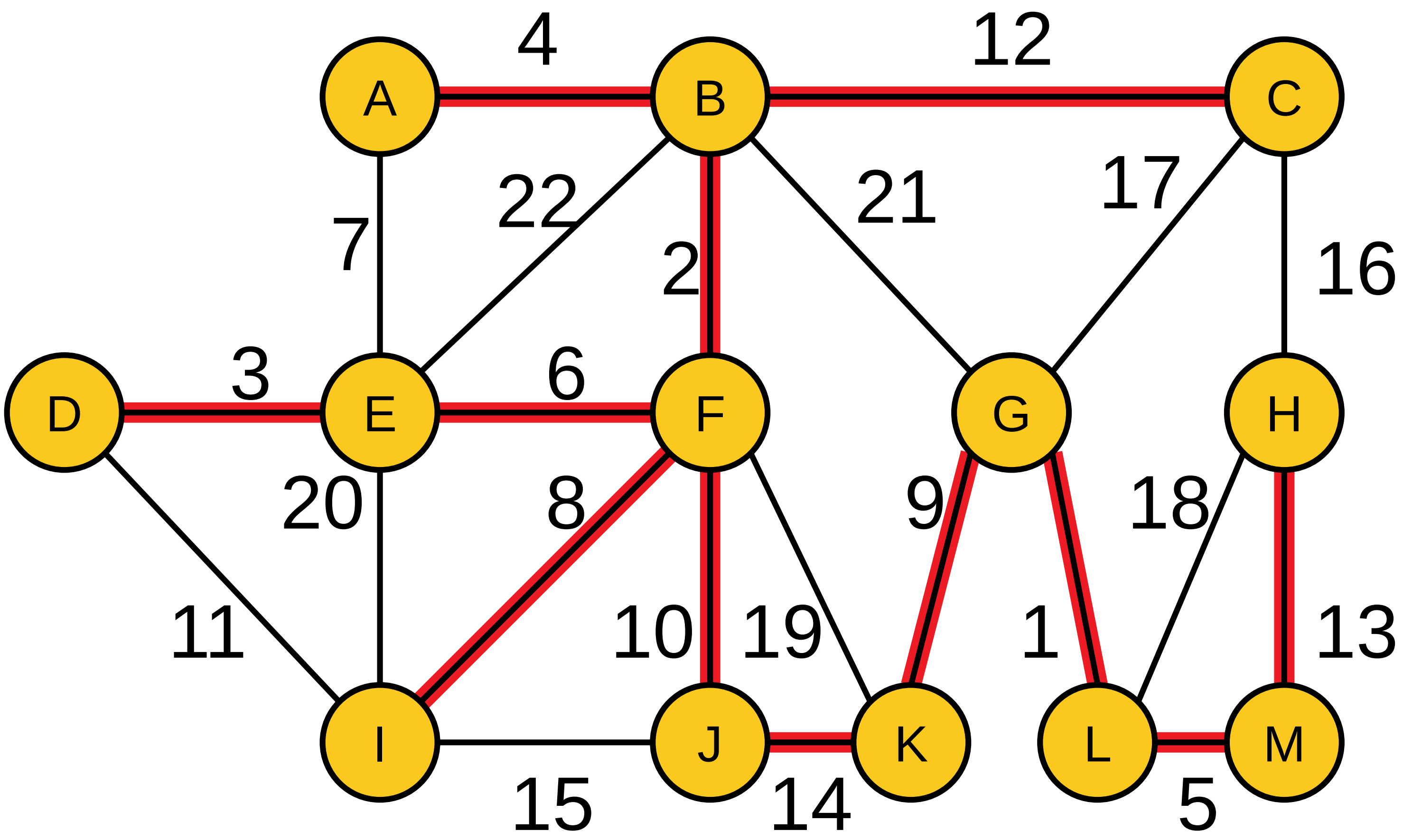
- Create forest F = (V,{}) with n components and no edge
- Put sorted edges (such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$) into set S
- While S non-empty and F not spanning:
  - delete cheapest edge from S
  - add it to F if no cycle is introduced

[1] Kruskal, J. B. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem". *Proceedings of the American Mathematical Society* **7**: 48–50. doi:10.1090/S0002-9939-1956-0078686-7

Kruskal's Algorithm: Example



Kruskal's Algorithm: Example



First question: how to implement the algorithm?

- sorting of edges needs O(|E| log |E|)

Algorithm

Create forest F = (V,{}) with n components and no edge
Put sorted edges (such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$) into set S
While S non-empty and F not spanning:
    delete cheapest edge from S
    add it to F if no cycle is introduced

simple

?

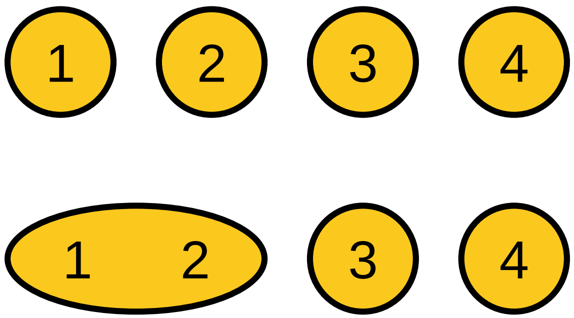
forest implementation:
Disjoint-set data structure

Disjoint-set Data Structure ("Union&Find")

Data structure: ground set 1...N grouped to disjoint sets

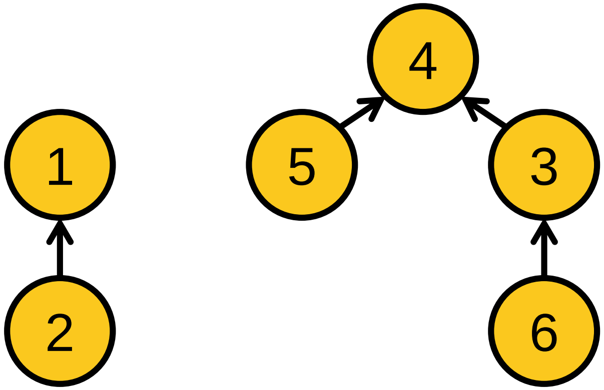
Operations:

- FIND(i): to which set ("tree") does i belong?
- UNION(i,j): union the sets of i and j!
  ("join the two trees of i and j")



Implemented as trees:

- UNION(T1, T2): hang root node of smaller tree under root node of larger tree (constant time), thus
- FIND(u): traverse tree from u to root (to return a representative of u's set) takes logarithmic time in total number of nodes



Implementation of Kruskal's Algorithm

Algorithm, rewritten with UNION-FIND:

- Create initial disjoint-set data structure, i.e. for each vertex v_i, store v_i as representative of its set
- Create empty forest F = {}
- Sort edges such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$
- for each edge e_i={u,v} starting from i=1:
  - if FIND(u) ≠ FIND(v): # no cycle introduced
    - $F = F \cup \{\{u,v\}\}$
    - UNION(u,v)
- return F

Back to Runtime Considerations

- Sorting of edges needs O(|E| log |E|)
- forest: **Disjoint-set data structure**
  - initialization: O(|V|)
  - log |V| to find out whether the minimum-cost edge {u,v} connects two sets (no cycle induced) or is within a set (cycle would be induced)
  - 2x FIND + potential UNION needs to be done O(|E|) times
  - total O(|E| log |V|)
- Overall: O(|E| log |E|)

Kruskal's Algorithm: Proof of Correctness

Two parts needed:

❶ Algo always produces a spanning tree

final F contains no cycle and is connected by definition ✓

❷ Algo always produces a *minimum* spanning tree

- argument by induction
- P: If *F* is forest at a given stage of the algorithm, then there is some minimum spanning tree that contains *F*.
- clearly true for F = (V, {})
- assume that P holds when new edge e is added to F and be T a MST that contains F
  - if e in T, fine
  - if e not in T: T + e has cycle C with edge f in C but not in F (otherwise e would have introduced a cycle in F)
    - now T – f + e is a tree with same weight as T (since T is a MST and f was not chosen to F)
    - hence T – f + e is MST including T + e (i.e. P holds)

✓

Another Greedy Algorithm for MST

- Another greedy approach to the MST problem is Prim's algorithm
- Somehow like the one of Kruskal but:
  - always keeps a tree instead of a forest
  - thus, take always the cheapest edge which connects to the current tree
- Runtime more or less the same for both algorithms, but analysis of Prim's algorithm a bit more involved because it needs (even) more complicated data structures to achieve it (hence not shown here)

Intermediate Conclusion

What we have seen so far:

- three problems where a greedy algorithm was optimal
  - money change
  - three circles in a triangle
  - minimum spanning tree (Kruskal’s and Prim’s algorithms)
- but also: greedy not always optimal
  - in particular for NP-hard problems

Obvious Question:

- when is greedy good?
- answer: matroids

Matroids

from Wikipedia:

"[...] a **matroid** is a structure that captures and generalizes the notion of linear independence in vector spaces."

Reminder: linear independence in vector spaces

again from Wikipedia:

"A set of vectors is said to be *linearly dependent* if one of the vectors in the set can be defined as a linear combination of the other vectors. If no vector in the set can be written in this way, then the vectors are said to be *linearly independent*."

Matroid: Definition

- Various equivalent definitions of matroids exist
- Here, we define a matroid via independent sets

Definition of a Matroid:

A *matroid* is a tuple $M = (E, \mathfrak{I})$ with

- E being the finite ground set and
- $\mathfrak{I}$ being a collection of (so-called) independent subsets of E satisfying these two axioms:
  - (I_1) if $X \subseteq Y$ and $Y \in \mathfrak{I}$ then $X \in \mathfrak{I}$,
  - (I_2) if $X \in \mathfrak{I}$ and $Y \in \mathfrak{I}$ and $|Y| > |X|$ then there exists an $e \in Y \setminus X$ such that $X \cup \{e\} \in \mathfrak{I}$.

Note: (I_2) implies that all *maximal independent sets* have the same cardinality (maximal independent = adding an item of E makes the set dependent)

Each maximal independent set is called a *basis* for M.

Example: Uniform Matroids

- A matroid $M = (E, \mathfrak{I})$ in which $\mathfrak{I} = \{X \subseteq E \colon |X| \leq k\}$ is called a *uniform matroid*.

- The bases of uniform matroids are the sets of cardinality k (in case $k \leq |E|$).

Example: Graphic Matroids

- Given a graph $G = (V, E)$, its corresponding *graphic matroid* is defined by $M = (E, \mathfrak{I})$ where $\mathfrak{I}$ contains all subsets of edges which are forests.

- If G is connected, the bases are the spanning trees of G.
- If G is unconnected, a basis contains a spanning tree in each connected component of G.

Matroid Optimization

Given a matroid $M = (E, \mathfrak{I})$ and a cost function $c: E \to \mathbb{R}$, the *matroid optimization problem* asks for an independent set S with the maximal total cost $c(S) = \sum_{e \in S} c(e)$.

- If all costs are non-negative, we search for a maximal cost basis.
- In case of a graphic matroid, the above problem is equivalent to the ***Maximum*** *Spanning Tree* problem (use Kruskal's algorithm, where the costs are negated, to solve it).

Greedy Optimization of a Matroid

Greedy algorithm on $M = (E, \mathfrak{I})$

- sort elements by their cost (w.l.o.g. $c(e_1) \geq c(e_2) \geq \cdots \geq c(e_{|M|})$)
- $S_0 = \{\}$, $k = 0$
- for $j = 1$ to $|E|$ do
  - if $S_k \cup e_j \in \mathfrak{I}$ then
    - $k = k + 1$
    - $S_k = S_{k-1} \cup e_j$
- output the sets $S_1, \ldots, S_k$ or $\max\{S_1, \ldots, S_k\}$

Theorem: The greedy algorithm on the independence system $M = (E, \mathfrak{I})$, which satisfies (I_1), outputs the optimum for any cost function iff M is a matroid.

Proof not shown here.

Exercise:
A Greedy Algorithm for the Knapsack Problem

Conclusions

I hope it became clear...

...what a greedy algorithm is

...that it not always results in the optimal solution

...but that it does if and only if the problem is a matroid