Introduction to Optimization
Greedy Algorithms

October 5, 2015

École Centrale Paris, Châtenay-Malabry, France

Dimo Brockhoff
INRIA Lille – Nord Europe

Course Overview

Date		Topic
Mon, 21.9.2015		Introduction
Mon, 28.9.2015	D	Basic Flavors of Complexity Theory
Mon, 5.10.2015	**D**	**Greedy algorithms**
Mon, 12.10.2015	D	Dynamic programming
Mon, 2.11.2015	D	Branch and bound/divide&conquer
Fri, 6.11.2015	D	Approximation algorithms and heuristics
Fri, 9.11.2015	C	Introduction to Continuous Optimization I
Fri, 13.11.2015	C	Introduction to Continuous Optimization II
Fri, 20.11.2015	C	Gradient-based Algorithms
Fri, 27.11.2015	C	End of Gradient-based Algorithms + Linear Programming
Fri, 4.12.2015	C	Stochastic Optimization and Derivative Free Optimization
Tue, 15.12.2015		Exam

all classes + exam last 3 hours (incl. a 15min break)

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!

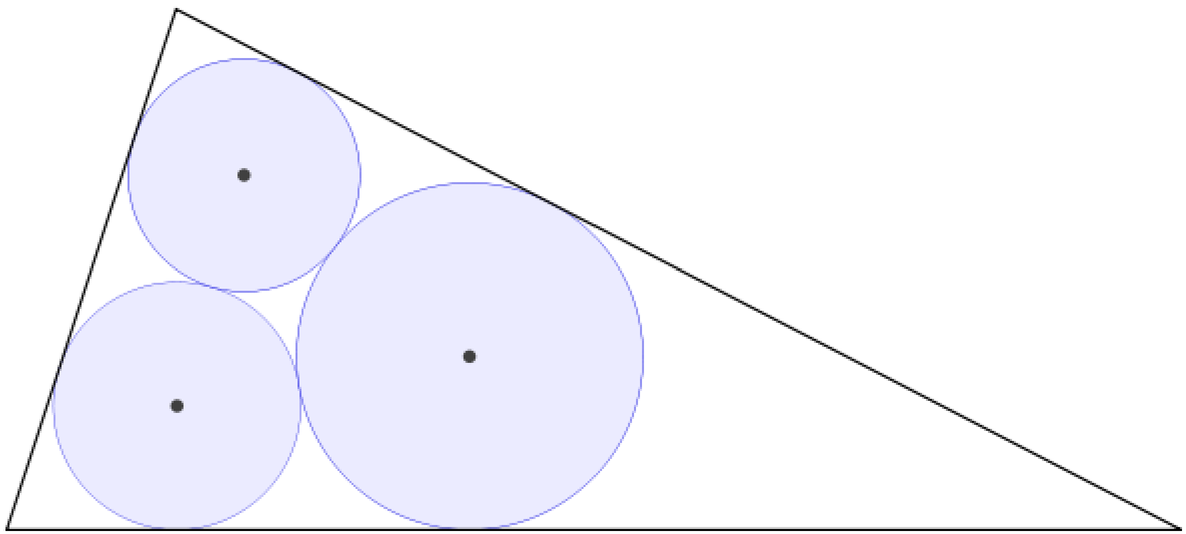
Related Problem:

finishing darts (from 501 to 0 with 9 darts)

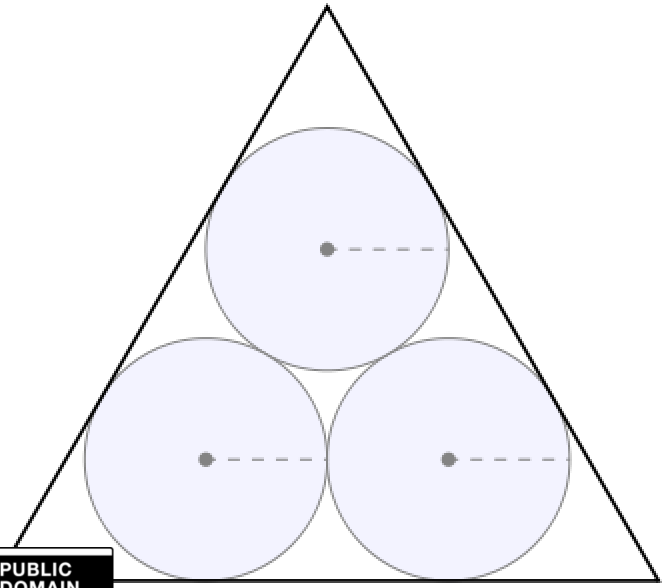
Example 2: Packing Circles in Triangles

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

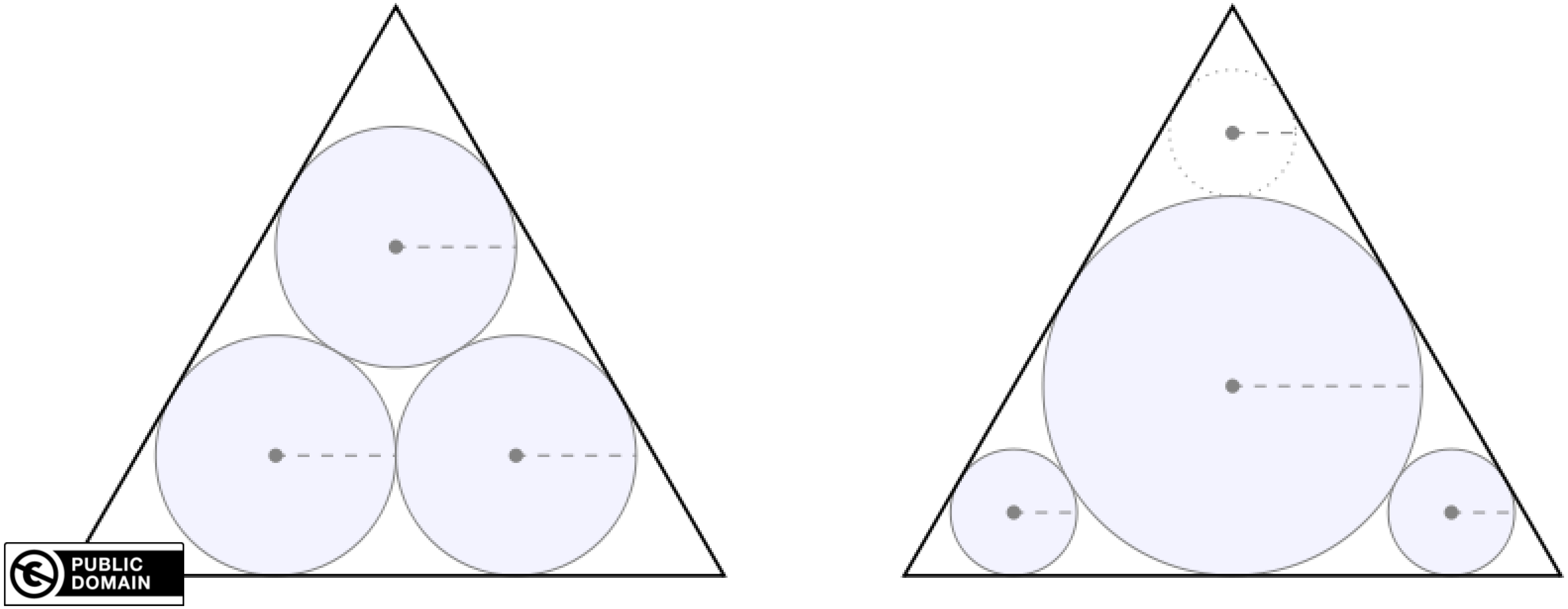


Example 2: Packing Circles in Triangles



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.

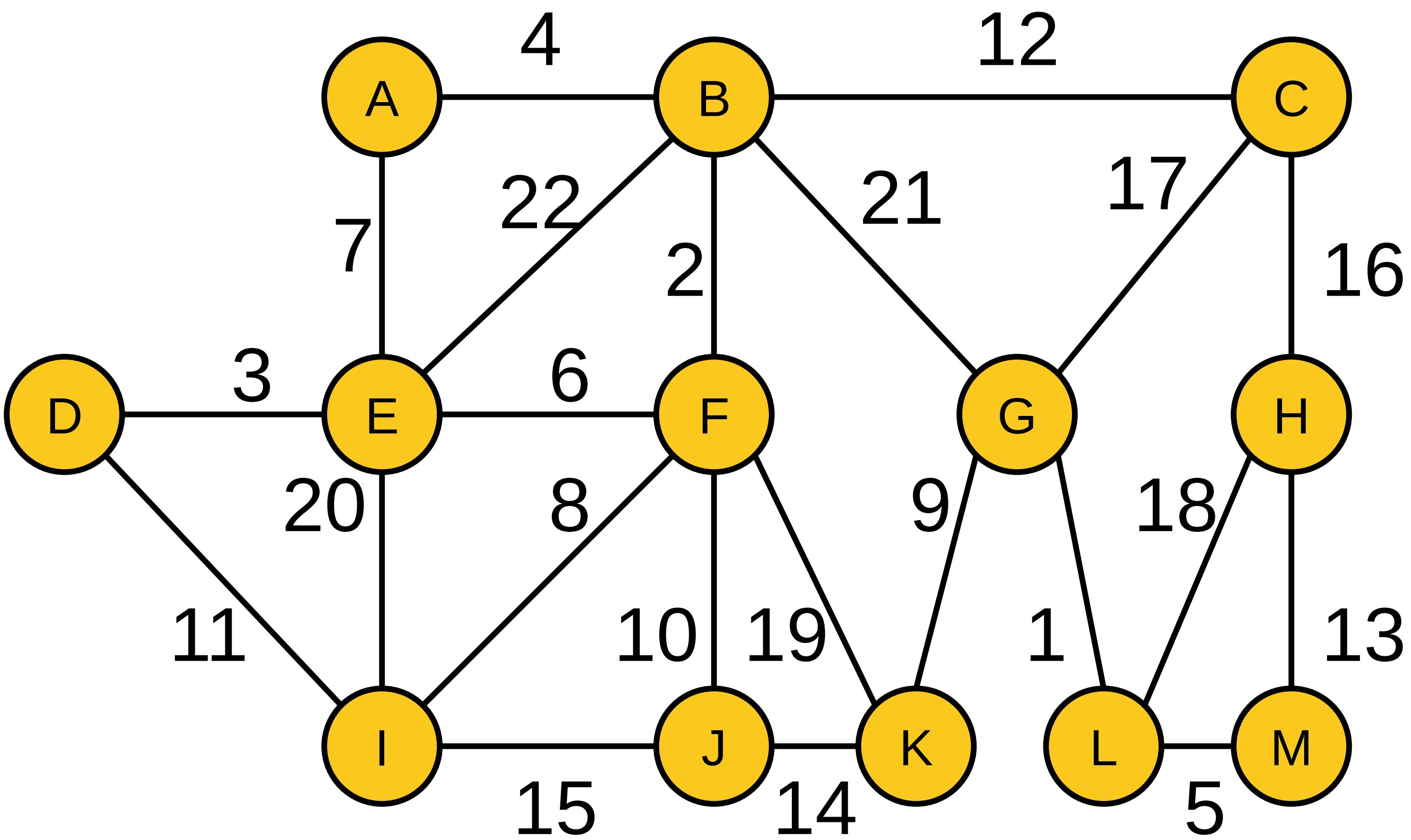
Kruskal's Algorithm

Algorithm, see [1]

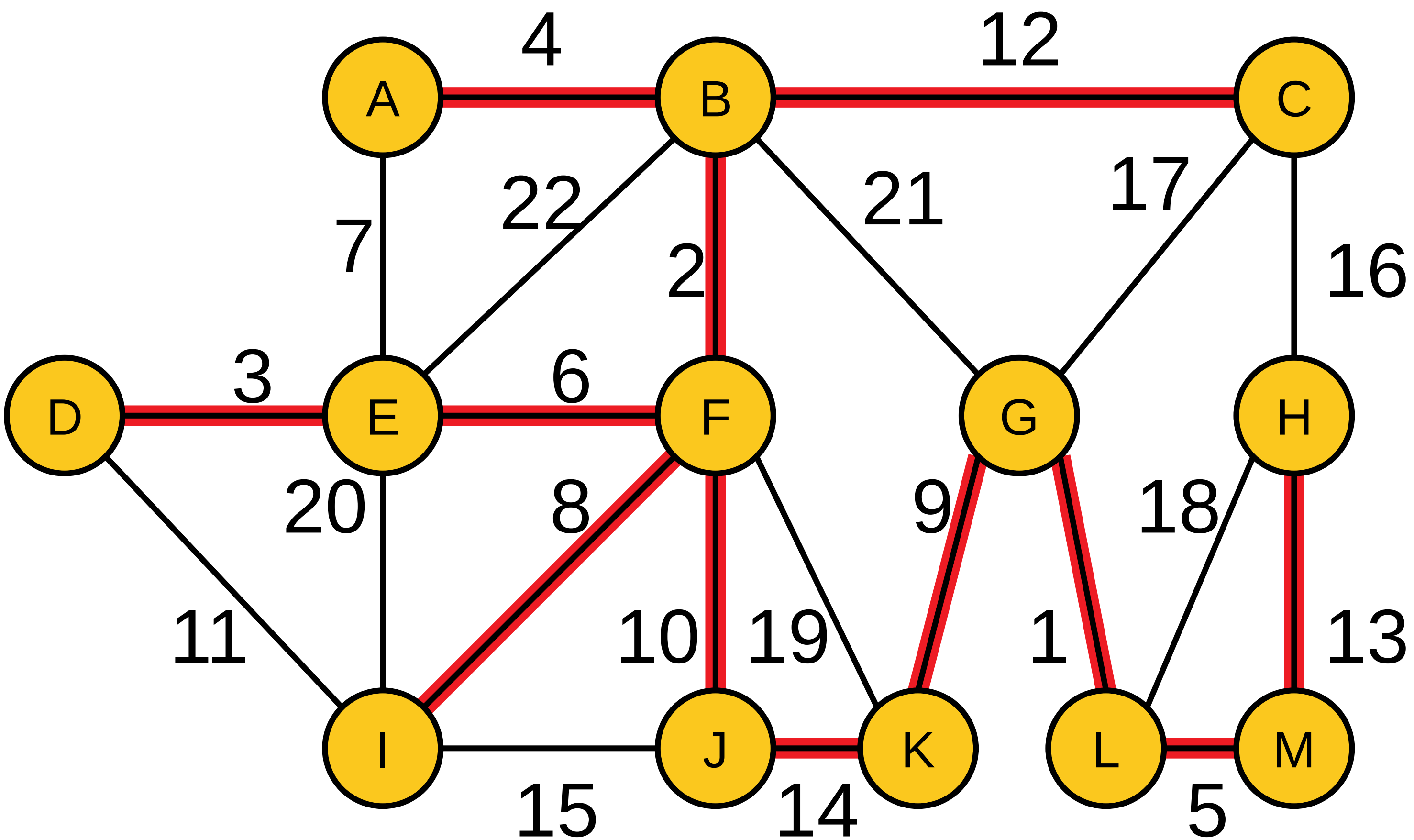
- Create forest F = (V,{}) with n components and no edge
- Put sorted edges (such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$) into set S
- While S non-empty and F not spanning:
  - delete cheapest edge from S
  - add it to F if no cycle is introduced

[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



Kruskal's Algorithm: Runtime Considerations

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. $w_1 < w_2 < \ldots < w_{|E|}$) 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

?

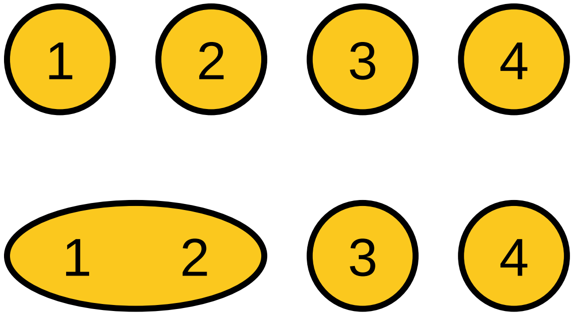
forest implementation:
Disjoint-set
data structure

Disjoint-set Data Structure (“Union&Find”)

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

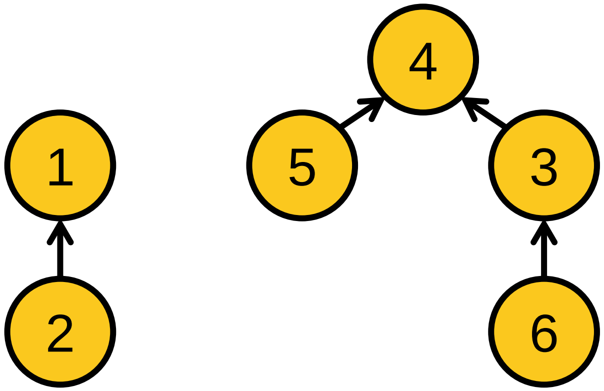
Operations:

- 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, $\mathcal{I}$) with

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

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 $M=(E, \mathcal{I})$ in which $\mathcal{I} = \{X \subseteq E: |X| \leq k\}$ 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, $\mathcal{I}$) where $\mathcal{I}$ 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, $\mathcal{I}$) and a cost function c: E $\rightarrow \mathbb{R}$, 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, $\mathcal{I}$)

- sort the elements by their cost s.t. w.l.o.g. $c(e_1) \geq c(e_2) \geq \ldots \geq e(e_{|M|})$
- $S_0 = \{\}$, k=0
- for j=1 to |E| do
  - if $S_k \cup e_j \in \mathcal{I}$ 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, $\mathcal{I}$), 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