Introduction to Optimization

Branch and Bound

November 4, 2016

École Centrale Paris, Châtenay-Malabry, France

Dimo Brockhoff

INRIA Lille – Nord Europe

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)

Greedy Algorithms (cont'd)

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

We will finally continue with the exercise "A Greedy Algorithm for the Knapsack Problem" after the branch and bound part

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.

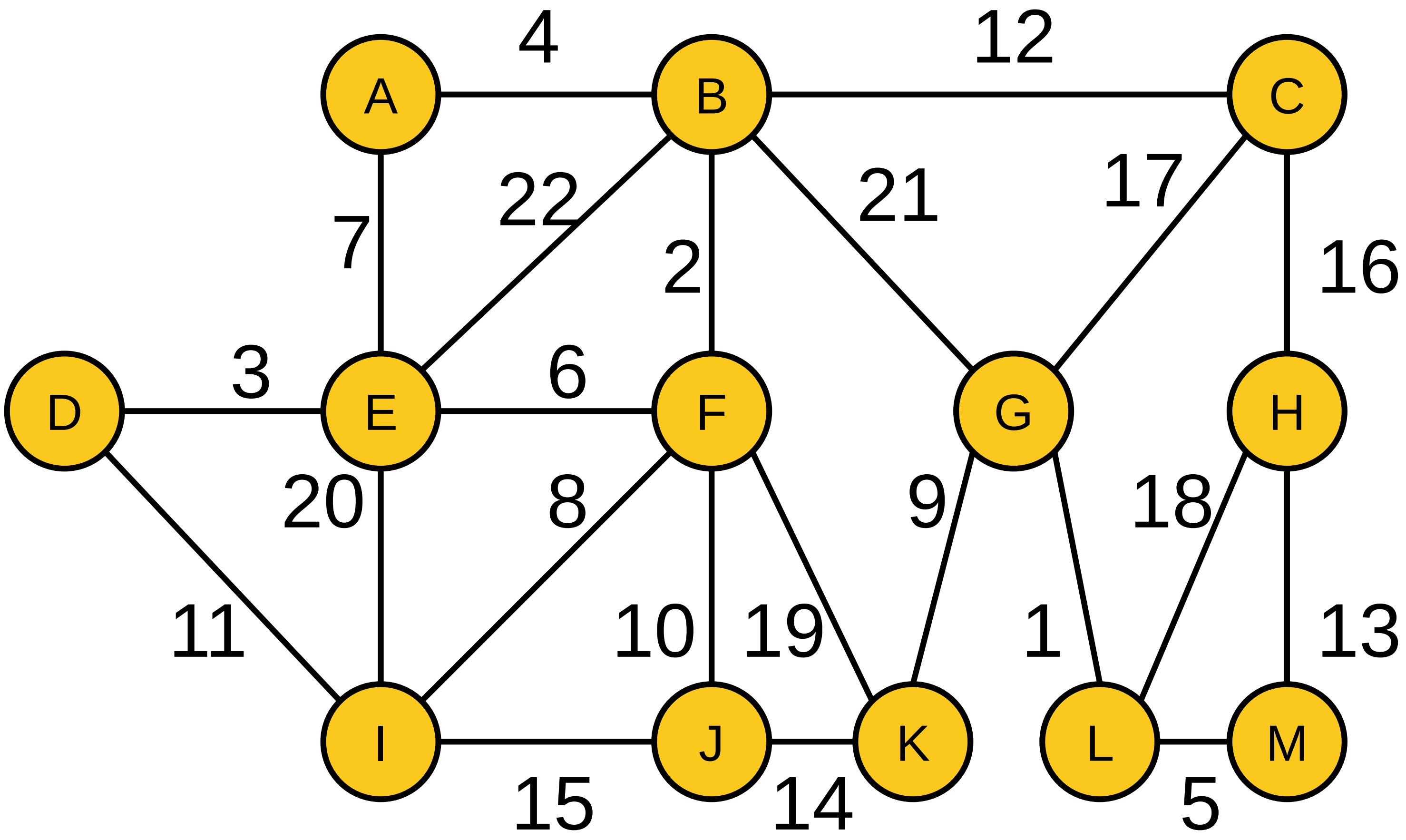
Kruskal's Algorithm

Algorithm, see [1]

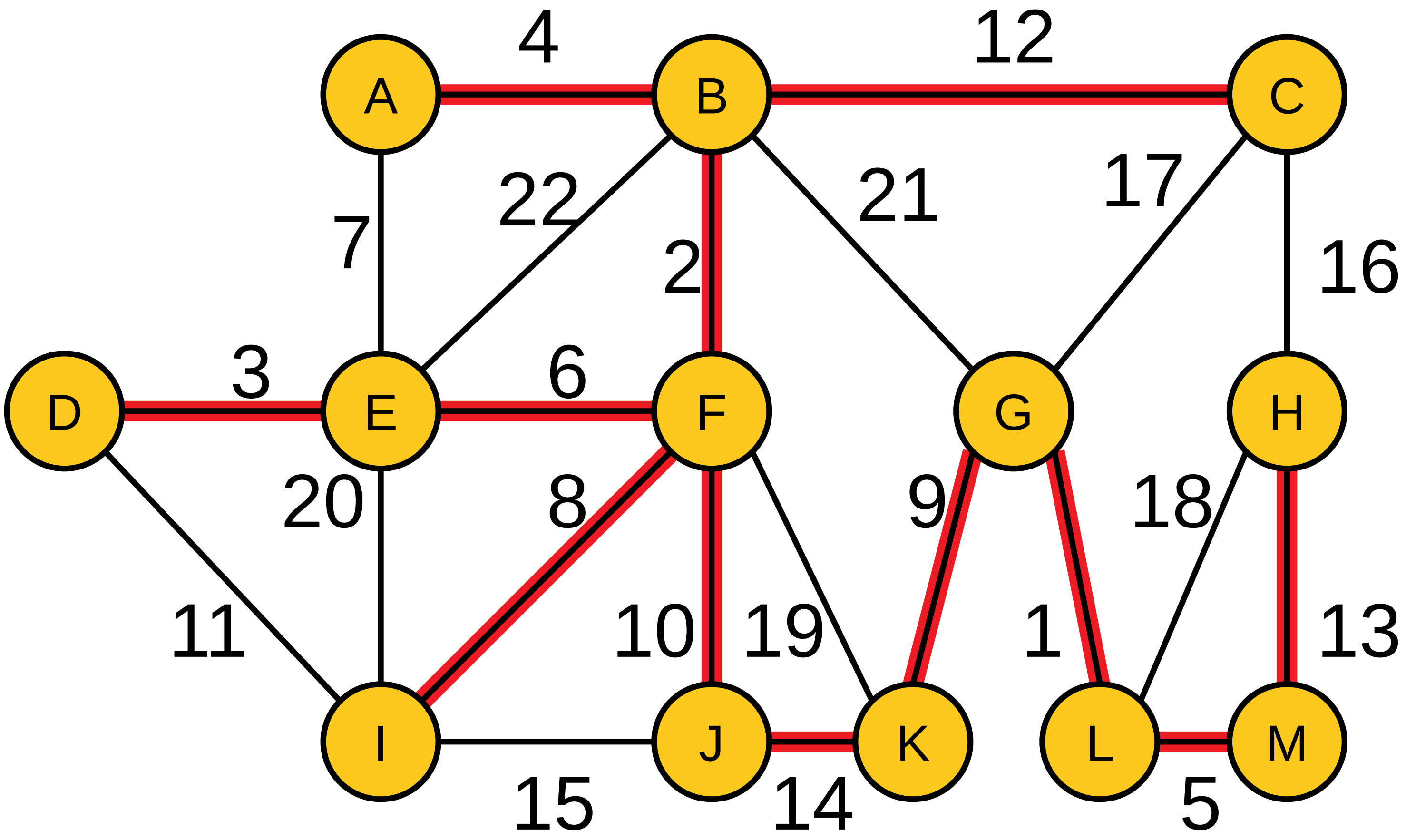
- Create forest F = (V,{}) with n components and no edge
- Put sorted edges (such that w.l.o.g. $w_1 < w_2 < \ldots < w_{|E|}$) into set S
- While S non-empty and F not spanning:
    - delete cheapest edge from S
    - add it to F if no cycle is introduced

[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



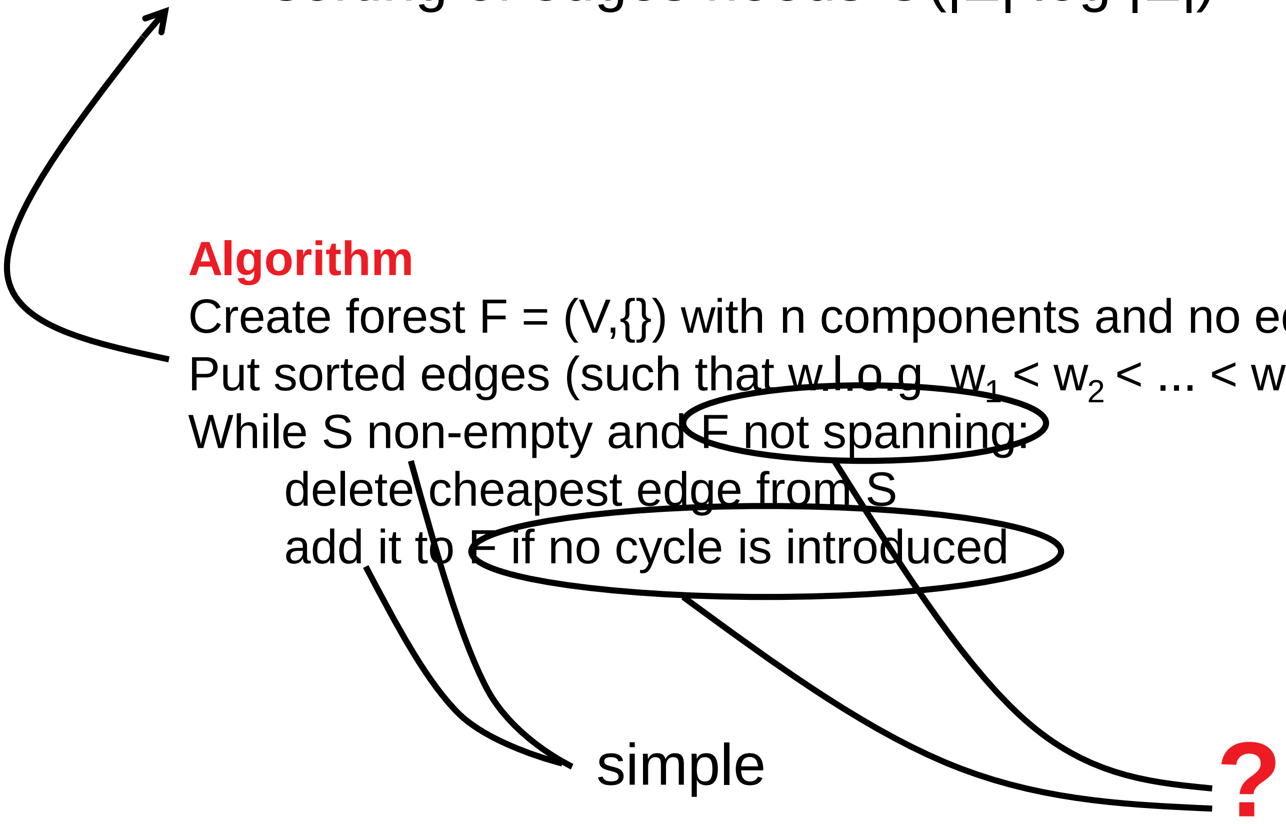
Kruskal's Algorithm: Example



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

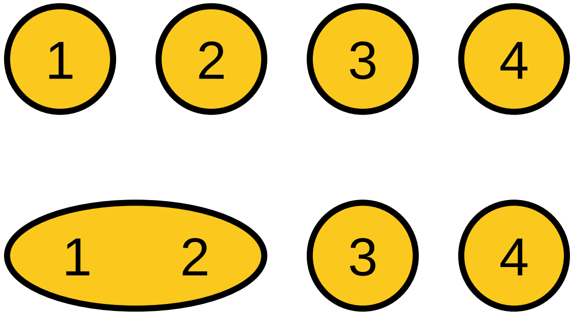
?

forest implementation:
Disjoint-set
data structure

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

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

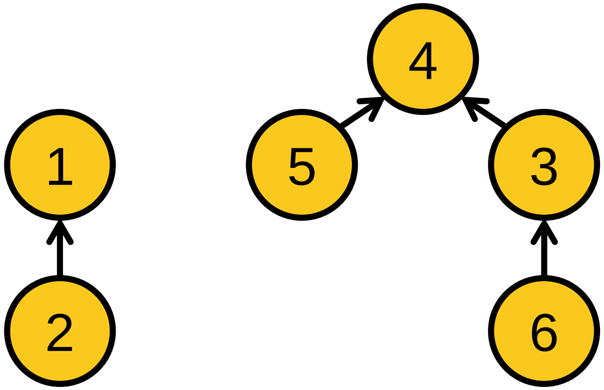
Operations:



- 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. $w_1 < w_2 < \ldots < w_{|E|}$
- 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 F + 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

Note: slides with blue background like the following have not been covered in the lecture and will therefore not been used in the exam.

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 $M = (E, \Im)$ 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.

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

Branch and Bound

Branch and Bound: General Ideas

Branch:

- Systematic enumeration of candidate solutions in a rooted tree
- Each tree node corresponds to a set of solutions; the whole search space on the root
- At each tree node, the corresponding subset of the search space is split into (non-overlapping) sub-subsets:
  - the optimum of the larger problem must be contained in at least one of the subproblems
- If tree nodes correspond to small enough subproblems, they are solved exhaustively

Bound:

- smart part: estimation of upper and lower bounds on the optimal function value achieved by solutions in the tree nodes
- the exploration of a tree node is stopped if a node's upper bound is already lower than the lower bound of an already explored node (assuming maximization)

Applying Branch and Bound

Needed for successful application of branch and bound:

- optimization problem
- finite set of solutions
- clear idea of how to split problem into smaller subproblems
- efficient calculation of the following modules:
  - upper bound calculation
  - lower bound calculation

Computing Bounds (Maximization Problems)

Assume w.l.o.g. maximization of f(x) here

Lower Bounds

- any actual feasible solution will give a lower bound (which will be exact if the solution is the optimal one for the subproblem)
- hence, sampling a (feasible) solution can be one strategy
- using a heuristic to solve the subproblem another one

Upper Bounds

- upper bounds can be achieved by solving a relaxed version of the problem formulations (i.e. by either loosening or removing constraints)

Note: the better/tighter the bounds, the quicker the branch and bound tree can be pruned

Properties of Branch and Bound Algorithms

- Exact, global solver
- Can be slow; only exponential worst-case runtime
    - due to the exhaustive search behavior if no pruning of the search tree is possible
- but might work well in some cases

Advantages:

- can be stopped if lower and upper bound are “close enough” in practice (not necessarily exact anymore then)
- can be combined with other techniques, e.g. “branch and cut” (not covered here)

Example Branching Decisions

0-1 problems:

- choose unfixed variable x_i
- one subproblem defined by setting x_i to 0
- one subproblem defined by setting x_i to 1

General integer problem:

- choose unfixed variable x_i
- choose a value c that x_i can take
- one subproblem defined by restricting $x_i \leq c$
- one subproblem defined by restricting $x_i > c$

Combinatorial Problems:

- branching on certain discrete choices, e.g. an edge/vertex is chosen or not chosen

The branching decisions are then induced as additional constraints when defining the subproblems.

Which Tree Node to Branch on?

Several strategies (again assuming maximization):

- choose the subproblem with highest upper bound
    - gain the most in reducing overall upper bound
    - if upper bound not the optimal value, this problem needs to be branched upon anyway sooner or later
- choose the subproblem with lowest lower bound
- simple DFS or BFS
- problem-specific approach most likely to be a good choice

4 Steps Towards a Branch and Bound Algorithm

Concrete steps when designing a branch and bound algorithm:

- How to split a problem into subproblems (“branching”)?
- How to compute upper bounds (assuming maximization)?
- Optional: how to compute lower bounds?
- How to decide which next tree node to split?

now: example of integer linear programming
example of knapsack problem (small exercise)

Application to ILPs

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

The ILP formalization covers many problems such as

- Traveling Salesperson Person (TSP)
- Vertex Cover and other covering problems
- Set packing and other packing problems
- Boolean satisfiability (SAT)

Ways of Solving an ILP

- Do not restrict the solutions to integers and round the solution found of the relaxed problem (=remove the integer constraints) by a continuous solver (i.e. solving the so-called *LP relaxation*)

  → no guarantee to be exact
- Exploiting the instance property of A being total unimodular:
  - feasible solutions are guaranteed to be integer in this case
  - algorithms for continuous relaxation can be used (e.g. the simplex algorithm)
- Using heuristic methods (typically without any quality guarantee)
  - we'll see these types of algorithms in one of the next lectures
- Using exact algorithms such as branch and bound

Branch and Bound for ILPs

Here, we just give an idea instead of a concrete algorithm...

- How to split a problem into subproblems (“branching”)?
- How to compute upper bounds (assuming maximization)?
- Optional: how to compute lower bounds?
- How to decide which next tree node to split?

Branch and Bound for ILPs

Here, we just give an idea instead of a concrete algorithm...

- How to compute upper bounds (assuming maximization)?
- How to split a problem into subproblems (“branching”)?
- Optional: how to compute lower bounds?
- How to decide which next tree node to split?

Branch and Bound for ILPs

How to compute upper bounds (assuming maximization)?

- drop the integer constraints and solve the so-called LP-relaxation
- can be done by standard LP algorithms such as **`scipy.optimize.linprog`** or Matlab's **`linprog`**

What's then?

- The LP has no feasible solution. Fine. Prune.
- We found an integer solution. Fine as well. Might give us a new lower bound to the overall problem.
- The LP problem has an optimal solution which is worse than the highest lower bound over all already explored subproblems. Fine. Prune.
- Otherwise: Branch on this subproblem: e.g. if optimal solution has $x_i=2.7865$, use $x_i \leq 2$ and $x_i \geq 3$ as new constraints

Branch and Bound for ILPs

How to split a problem into subproblems (“branching”)?

  - mainly needed if the solution of the LP-relaxation is not integer
  - branch on a variable which is rational

Not discussed here in depth due to time:

- Optional: how to compute lower bounds?
- How to decide which next tree node to split?
  - seems to be good choice: subproblem with largest upper bound of LP-relaxation

Branch and Bound for the 0-1 Knapsack Problem

How would you implement a branch-and-bound algorithm for the 0-1 knapsack problem?

what are the subproblems?
how to split a problem?

how to compute upper bounds?

how to compute lower bounds?

Branch and Bound for the Knapsack Problem

Ideas:

- define subproblems by choosing one variable and setting it to either 0 or 1 (those fixed values are then ensured by additional constraints in the problem formulation)
- for computing upper bounds for each subproblem, we can relax the binary values constraints and use a greedy algorithm that can pack items “partially”
- good lower bounds can be computed by a simple greedy algorithm (see today’s exercise)

Conclusions

I hope it became clear...

...what the basic algorithm design ideas of branch and bound are

...and for which problem types it is supposed to be suitable

back to the exercise:
A Greedy Algorithm for the Knapsack Problem

`http://researchers.lille.inria.fr/ ~brockhof/optimizationSaclay/`