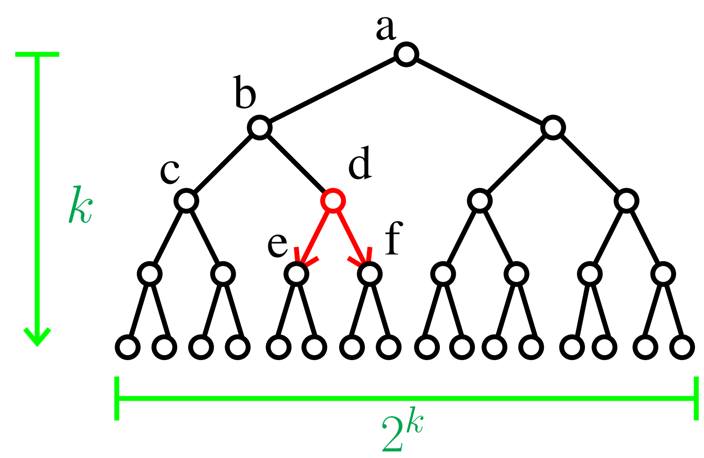
Coping with NP-completeness 1

IN3130 Algorithms and Complexity

Coping with Intractability

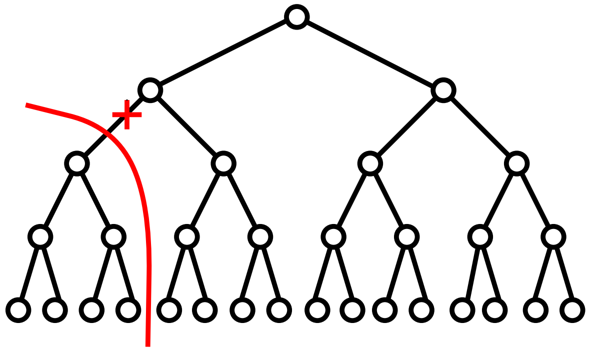
Branch-and-Bound

Branch:



Leaf nodes = possible solutions

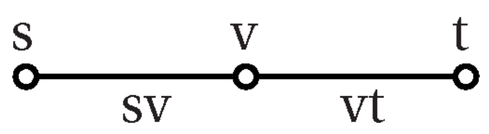
Bound:



- Bactracking
- Pruning ('avskjæring')

Dynamic Programming

- Building up a solution from solutions from subproblems
- Principle: Every part of an optimal solution must be optimal.



Restricting

- **Idea:** Perhaps the hard instances don't arise in practice?
- Often **restricted versions** of intractable problems can be solved efficiently.

Some examples:

- CLIQUE on graphs with edge degrees bounded by constant is in $\mathcal{P}$:
  const. $C \Rightarrow \binom{n}{C} = \mathcal{O}\left(n^C\right)$ is a polynomial!
- Perhaps the input **graphs** are
  — planar
  — sparse
  — have limited degrees
  — ...
- Perhaps the input **numbers** are
  — small
  — limited
  — ...

Pseudo-polynomial algorithms

Def. 1 *Let I be an instance of problem L, and let MAXINT(I) be (the value of) the largest integer in I. An algorithm which solves L in time which is polynomial in |I| and MAXINT(I) is said to be a **pseudo-polynomial algorithm** for L.*

Note: If MAXINT(I) is a constant or even a polynomial in |I| for all I $\in L$, then a pseudo-polynomial algorithm for L is also a polynomial algorithm for L.

Example: 0-1 KNAPSACK

In 0-1 KNAPSACK we are given integers $w_1, w_2, \ldots, w_n$ and K, and we must decide whether there is a subset S of $\{1, 2, \ldots, n\}$ such that $\sum_{j \in S} w_j = K$. In other words: Can we put a subset of the integers into our knapsack such that the knapsack sums up to exactly K, under the restriction that we include any w_i at most one time in the knapsack.

Note: This decision version of 0-1 KNAPSACK is essentially SUBSET SUM.

0-1 KNAPSACK can be solved by dynamic programming. **Idea:** Going through all the w_i one by one, maintain an (ordered) set M of all sums ($\leq K$) which can be computed by using some subset of the integers seen so far.

Algorithm DP

```
1.Let M_0 := {0}.
2.For j = 1,2,...,n do:
   Let M_j := M_{j-1}.
   For each element u ∈ M_{j-1}:
     Add v = w_j + u to M_j if v ≤ K and
      v is not already in M_j.
3.Answer 'Yes' if K ∈ M_n, 'No'
otherwise.
```

Example: Consider the instance with w_i's $11, 18, 24, 42, 15, 7$ and $K = 56$. We get the following M_i-sets:

$M_0 : \{0\}$
$M_1 : \{0, 11\} \quad (0 + 11 = 11)$
$M_2 : \{0, 11, 18, 29\} \quad (0 + 18 = 18, 11 + 18 = 29)$
$M_3 : \{0, 11, 18, 24, 29, 35, 42, 53\}$
$M_4 : \{0, 11, 18, 24, 29, 35, 42, 53\}$
$M_5 : \{0, 11, 15, 18, 24, 26, 29, 33,$
$35, 39, 42, 44, 50, 53\}$
$M_6 : \{0, 7, 11, 15, 18, 22, 24, 25, 26, 29, 31, 33,$
$35, 36, 39, 40, 42, 44, 46, 49, 50, 51, 53\}$

Theorem 1 *DP is a pseudo-polynomial algorithm. The running time of DP is* $\mathcal{O}(nK \log K)$.

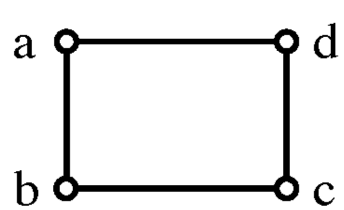
Proof: MAXINT(I)$= K \ldots$

Strong $\mathcal{NP}$-completeness

Def. 2 *A problem which has no pseudo-polynomial algorithm unless $\mathcal{P} = \mathcal{NP}$ is said to be* ***$\mathcal{NP}$-complete in the strong sense*** *or* ***strongly $\mathcal{NP}$-complete***.

Theorem 2 TSP *is strongly $\mathcal{NP}$-complete.*

Proof: In the standard reduction HAM $\propto$TSP the only integers are $1, 2$ and n, so MAXINT(I)$= n$. Hence a pseudo-polynomial algorithm for TSP would solve HAMILTONICITY in polynomial time (via the standard reduction).



$\propto$

	a	b	c	d
a	2	1	2	1
b	1	2	1	2
c	2	1	2	1
d	1	2	1	2

$$K = n(= 4)$$

Alternative approaches to algorithm design and analysis

- **Problem:** Exhaustive search gives typically $\mathcal{O}(n!) \approx \mathcal{O}(n^n)$-algorithms for $\mathcal{NP}$-complete problems.

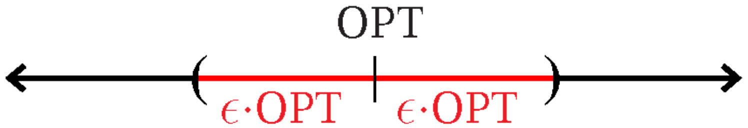
- So we need to get around the **worst case / best solution** paradigm:

— worst-case $\rightarrow$ average-case analysis

— best solution $\rightarrow$ approximation

— best solution $\rightarrow$ randomized algorithms

Approximation

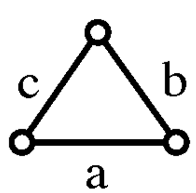


Def. 3 *Let L be an optimization problem. We say that algorithm M is a **polynomial-time ϵ-approximation algorithm** for L if M runs in polynomial time and there is a constant $\epsilon \geq 0$ such that M is guaranteed to produce, for all instances of L, a solution whose cost is within an ϵ-neighborhood from the optimum.*

Note 1: Formally this means that the **relative error** $\frac{|t_M(n) - \mathrm{OPT}|}{\mathrm{OPT}}$ must be less than or equal to the constant ϵ.

Note 2: We are still looking at the worst case, but we don't require the very best solution any more.

Example: TSP with triangle inequality has a polynomial-time approximation algorithm.

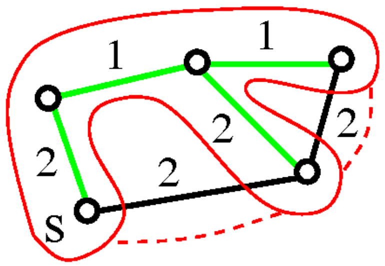


$c \leq a + b$

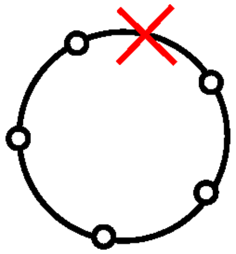
Algorithm TSP-$\triangle$:

Phase I: Find a minimum spanning tree.

Phase II: Use the tree to create a tour.



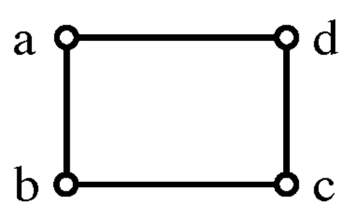
The cost of the produced solution can not be more than $2 \cdot$OPT, otherweise the OPT tour (minus one edge) would be a more minimal spanning tree itself. Hence $\epsilon = 1$.



Opt. tour

Theorem 3 TSP *has no polynomial-time* ϵ*-approximation algorithm for any* ϵ *unless* $\mathcal{P}=\mathcal{NP}$.

Proof:
Idea: Given ϵ, make a reduction from HAMILTONICITY which has only **one** solution within the ϵ-neighborhood from OPT, namely the optimal solution itself.



$\propto$

	a	b	c	d
a	2+ϵ n	1	2+ϵ n	1
b	1	2+ϵ n	1	2+ϵ n
c	2+ϵ n	1	2+ϵ n	1
d	1	2+ϵ n	1	2+ϵ n

$$K = n(= 4)$$

The **error** resulting from picking a non-edge is: Approx.solutin - OPT =
$(n - 1 + 2 + \epsilon n) - n = (1 + \epsilon)n > \epsilon n$

Hence a polynomial-time ϵ-approximation algorithm for TSP combined with the above reduction would solve HAMILTONICITY in polynomial time.

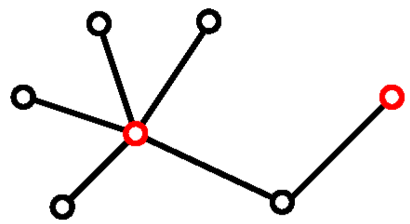
Example: Vertex Cover

- **Heuristics** are a common way of dealing with intractable (optimization) problems in practice.
- Heuristics differ from algorithms in that they have no performance guarantees, i.e. they don't always find the (best) solution.

A greedy heuristic for Vertex Cover-opt.:

Heuristic VC-H1:

```
Repeat until all edges are covered:
   1.Cover highest-degree vertex v;
   2.Remove v (with edges) from
     graph;
```



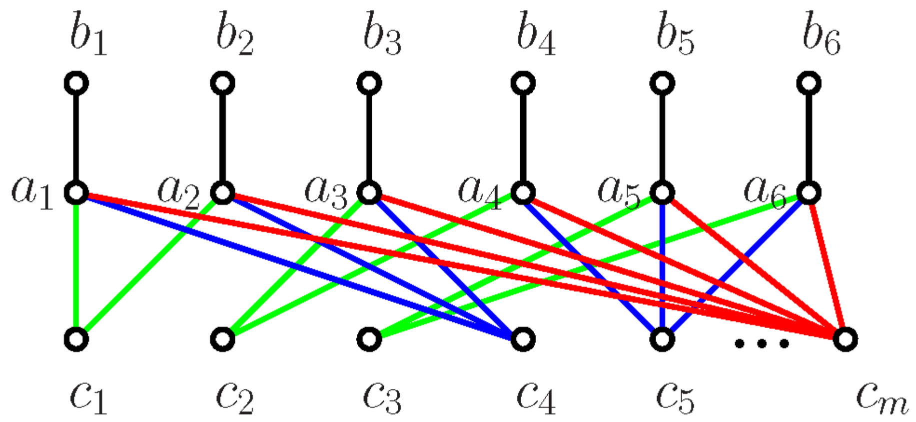
Theorem 4 *The heuristic VC-H1 is not an ϵ-approximation algorithm for* Vertex Cover*-opt. for any fixed ϵ.*

Proof:

Show a **counterexample**, i.e. cook up an instance where the heuristic performs badly.

Counterexample:

- A graph with nodes $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$.
- Node b_i is only connected to node a_i.
- A bunch of c-nodes connected to a-nodes in the following way:
  - Node c_1 is connected to a_1 and a_2. Node c_2 is connected to a_3 and a_4, etc.
  - Node $c_{n/2+1}$ is connected to a_1, a_2 and a_3. Node $c_{n/2+2}$ is connected to a_4, a_5 and a_6, etc.
  - ...
  - Node c_{m-1} is connected to $a_1, a_2, \ldots a_{n-1}$.
  - Node c_m is connected to all a-nodes.



- The optimal solution OPT requires n guards (on all a-nodes).
- VC-H1 first covers all the c-nodes (starting with c_m) before covering the a-nodes.
- The number of c-nodes are of order $n \log n$.
- Relative error for VC-H1 on this instance:

$$\frac{|\text{VC-H1}| - |\text{OPT}|}{|\text{OPT}|} = \frac{(n \log n + n) - n}{n}$$
$$= \frac{n \log n}{n} = \log n \neq \epsilon$$

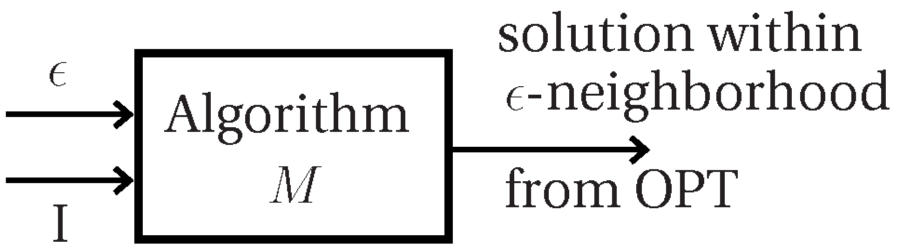
- The relative error **grows as a function of n**.

Heuristic VC-H2:

```
Repeat until all edges are covered:
   1.Pick an edge e;
   2.Cover and remove
     both endpoints of e.
```

- Since at least one endpoint of every edge must be covered, $|\text{VC-H2}| \leq 2 \cdot |\text{OPT}|$.
- So VC-H2 is a polynomial-time ϵ-approximation algorithm for VC with $\epsilon = 1$.
- Surpisingly, this "stupid-looking" algorithm is the best (worst case) approximation algorithm known for VERTEX COVER-opt.

Polynomial-time approximation schemes (PTAS)



Running time of M is $\mathcal{O}(P_\epsilon(|I|))$ where $P_\epsilon(n)$ is a polynomial in n and also a function of ϵ.

Def. 4 *M is a* ***polynomial-time approximation scheme (PTAS)*** *for optimization problem L if given an instance I of L and value $\epsilon > 0$ as input*

1. *M produces a solution whose cost is within an ϵ-neigborhood from the optimum (OPT) and*

2. *M runs in time which is bounded by a polynomial (depending on ϵ) in $|I|$.*

M is a ***fully polynomial-time approximation scheme (FPTAS)*** *if it runs in time bounded by a polynomial in $|I|$ and $1/\epsilon$.*

Example: 0-1 KNAPSACK-optimization has a FPTAS.

0-1 KNAPSACK-optimization

Instance: $2n+1$ integers: Weights $w_1, \ldots, w_n$ and costs $c_1, \ldots, c_n$ and maximum weight K.

Question: Maximize the total cost

$$\sum_{j=1}^{n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} w_j x_j \leq K \text{ and } x_j = 0, 1$$

Image: We want to maximize the total value of the items we put into our knapsack, but the knapsack cannot have total weight more than K and we are only allowed to bring one copy of each item.

Note: Without loss of generality, we shall assume that all individual weights w_j are $\leq K$.

0-1 KNAPSACK-opt. can be solved in pseudo-polynomial time by dynamic programming. **Idea:** Going through all the items one by one, maintain an (ordered) set M of pairs (S, C) where S is a subset of the items (represented by their indexes) seen so far, such that S is the "lightest" subset having total cost equal C.

Algorithm DP-OPT

1.Let $M_0 := \{(\emptyset, 0)\}$.

2.For $j = 1, 2, \ldots, n$ do steps (a)-(c):

(a) Let $M_j := M_{j-1}$.

(b) For each elem.(S, C) of M_{j-1}:
If $\sum_{i \in s} w_i + w_j \leq K$, then add
$(S \cup \{j\}, C + c_j)$ to M_j.

(c) Examine M_j for pairs of
elements (S, C) and (S', C)
with the same 2nd component.
For each such pair, delete
(S', C) if $\sum_{i \in s'} w_i \geq \sum_{i \in S} w_i$
and delete (S, C) otherwise.

3.The optimal solution is S where (S, C)
is the element of M_n having the larges
second component.

- The running time of DP-OPT is $\mathcal{O}\left(n^2 C_m \log(n C_m W_m)\right)$ where C_m and W_m are the largest cost and weight, respectively.

Example: Consider the following instance of 0-1 KNAPSACK-opt.

j	1	2	3	4
w_j	1	1	3	2
c_j	6	11	17	3

$K = 5$

Running the DP-OPT algorithm results in the following sets:

$$M_0 = \{(\emptyset, 0)\}$$
$$M_1 = \{(\emptyset, 0), (\{1\}, 6)\}$$
$$M_2 = \{(\emptyset, 0), (\{1\}, 6), (\{2\}, 11), (\{1, 2\}, 17)\}$$
$$M_3 = \{(\emptyset, 0), (\{1\}, 6), (\{2\}, 11), (\{1, 2\}, 17), (\{1, 3\}, 23), (\{2, 3\}, 29), (\{1, 2, 3\}, 34)\}$$
$$M_4 = \{(\emptyset, 0), (\{4\}, 3), (\{1\}, 6), (\{1, 4\}, 9), (\{2\}, 11), (\{2, 4\}, 14), (\{1, 2\}, 17), (\{1, 2, 4\}, 20), (\{1, 3\}, 23), (\{2, 3\}, 29), (\{1, 2, 3\}, 34)\}$$

Hence the optimal subset is $\{1, 2, 3\}$ with $\sum_{j \in S} c_j = 34$.

The FTPAS for 0-1 KNAPSACK-optimization combines the DP-OPT algorithm with rounding-off of input values:

j	1	2	3	4	5	6	7
w_j	4	1	2	3	2	1	2
c_j	299	73	159	221	137	89	157

$K = 10$

The optimal solution $S = \{1, 2, 3, 6, 7\}$ gives $\sum_{j \in S} c_j = 777$.

j	1	2	3	4	5	6	7
w_j	4	1	2	3	2	1	2
$\bar{c}_j$	290	70	150	220	130	80	150

$K = 10$

The best solution, given the trunctation of the last digit in all costs, is $S' = \{1, 3, 4, 6\}$ with $\sum_{j \in S'} c_j = 740$.

Algorithm APPROX-DP-OPT

- Given an instance I of 0-1 KNAPSACK-opt and a number t, truncate (round off downward) t digits of each cost c_j in I.
- Run the DP-OPT algorithm on this truncated instance.
- Give the answer as an approximation of the optimal solution for I.

Idea:

- Truncating t digits of all costs, reduces the number of possible "cost sums" by a factor exponential in t. This implies that **the running time drops exponentially**.
- **Truncating error** relative to reduction in instance size is "**exponentially small**":

$$C_m = 53501\,\underbrace{87959}_{\substack{\text{half of length}\\ \text{but only } 10^{-5} \text{ of}\\ \text{precision}}}$$

Theorem 5 *APPROX-DP-OPT is a FPTAS for 0-1 KNAPSACK-opt.*

Proof: Let S and S' be the optimal solution of the original and the truncated instance of 0-1 KNAPSACK-opt., respectively. Let c_j and $\bar{c}_j$ be the original and truncated version of the cost associated with element j. Let t be the number of truncated digits. Then

$$\sum_{j \in S} c_j \overset{(1)}{\geq} \sum_{j \in S'} c_j \overset{(2)}{\geq} \sum_{j \in S'} \bar{c}_j \overset{(3)}{\geq} \sum_{j \in S} \bar{c}_j$$

$$\overset{(4)}{\geq} \sum_{j \in S} (c_j - 10^t) \overset{(5)}{\geq} \sum_{j \in S} c_j - n \cdot 10^t$$

1. because S is a optimal solution
2. because we round off downward ($\bar{c}_j \leq c_j$ for all j)
3. because S' is a optimal solution for the truncated instance
4. because we truncate t digits
5. because S has at most n elements

This means that the have an upper bound on the **error**:

$$\sum_{j \in S} c_j - \sum_{j \in S'} c_j \leq n \cdot 10^t$$

- Running time of DP-OPT is $\mathcal{O}\left(n^2 C_m \log(nC_m W_m)\right)$ where C_m and W_m are the largest cost and weight, respectively.

- Running time of APPROX-DP-OPT is $\mathcal{O}\left(n^2 C_m \log(nC_m W_m) 10^{-t}\right)$ because by truncating t digits we have reduced the number of possible "cost sums" by a factor 10^t.

- Relative error ϵ is

$$\frac{\sum_{j\in S} c_j - \sum_{j\in S'} c_j}{\sum_{j\in S} c_j} \overset{(1)}{\leq} \frac{n \cdot 10^t}{c_m} \triangleq \epsilon$$

  1. because our assumption that each individual weight w_j is $\leq K$ ensures that $\sum_{j\in S} c_j \geq C_m$ (the item with cost C_m always fits into an empty knapsack).

- Given any $\epsilon > 0$, by truncating $t = \lfloor \log_{10} \frac{\epsilon \cdot c_m}{n} \rfloor$ digits APPROX-DP-OPT is an ϵ-approximation algorihtm for 0-1 KNAPSACK-opt with running time $\mathcal{O}\left(\frac{n^3 \log(nC_m W_m)}{\epsilon}\right)$.