Approximation algorithms

Some optimisation problems are “hard”, little chance of finding poly-time algorithm that computes **optimal** solution

- **largest** clique
- **smallest** vertex cover
- **largest** independent set

But: We can calculate a **sub-optimal** solution in poly time.

- **pretty large** clique
- **pretty small** vertex cover
- **pretty large** independent set

Approximation algorithms compute **near-optimal** solutions.

Known for thousands of years. For instance, approximations of value of π; some engineers still use 4 these days `:-)`

Consider **optimisation problem**.

Each potential solution has **positive cost**, we want **near-optimal** solution.

Depending on problem, optimal solution may be one with

- **maximum possible cost** (maximisation problem), like maximum clique,
- or one with **minimum possible cost** (minimisation problem), like minimum vertex cover.

Algorithm has **approximation ratio** of $\rho(n)$, if for any input of size n, the cost C of its solution is **within factor** $\rho(n)$ of cost of optimal solution C^*, i.e.

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

Maximisation problems:

- $0 < C \leq C^*$,
- C^*/C gives factor by which optimal solution is better than approximate solution (note: $C^*/C \geq 1$ and $C/C^* \leq 1$).

Minimisation problems:

- $0 < C^* \leq C$,
- C/C^* gives factor by which optimal solution is better than approximate solution (note $C/C^* \geq 1$ and $C^*/C \leq 1$).

Approximation ratio is **never** less than one:

$$\frac{C}{C^*} < 1 \;\Rightarrow\; \frac{C^*}{C} > 1$$

Approximation Algorithm

An algorithm with guaranteed approximation ration of $\rho(n)$ is called a $\rho(n)$-**approximation algorithm**.

A 1-approximation algorithm is optimal, and the larger the ratio, the worse the solution.

- For many $\mathcal{NP}$-complete problems, **constant-factor approximations exist** (i.e. computed clique is always at least half the size of maximum-size clique),
- sometimes in best known approx ratio grows with n,
- and sometimes even proven lower bounds on ratio (*for every approximation alg, the ratio is at least this and that, unless* $\mathcal{P} = \mathcal{NP}$).

Approximation Scheme

Sometimes the approximation ratio improves when spending more computation time.

An **approximation scheme** for an optimisation problem is an approximation algorithm that takes as input an instance **plus** a parameter $\epsilon > 0$ s.t. for any fixed ϵ, the scheme is a $(1 + \epsilon)$-approximation (*trade-off*).

PTAS and FPTAS

A scheme is a **poly-time approximation scheme** (PTAS) if for any fixed $\epsilon > 0$, it runs in time polynomial in input size.

Runtime can increase **dramatically** with decreasing ϵ, consider $T(n) = n^{2/\epsilon}$.

n	ϵ / $T(n)$	2 / n	1 / n^2	$1/2$ / n^4	$1/4$ / n^8	$1/100$ / n^{200}
10^1		10^1	10^2	10^4	10^8	10^{200}
10^2		10^2	10^4	10^8	10^{16}	10^{400}
10^3		10^3	10^6	10^{12}	10^{24}	10^{600}
10^4		10^4	10^8	10^{16}	10^{32}	10^{800}

We want: if ϵ **decreases** by constant factor, then running time **increases by at most** some other constant factor, i.e., running time is polynomial in n **and** $1/\epsilon$. Example: $T(n) = (2/\epsilon) \cdot n^2$, $T(n) = (1/\epsilon)^2 \cdot n^3$.

Such a scheme is called a **fully polynomial-time approximation scheme** (FPAS).

Example 1: Vertex cover

Problem: given graph $G = (V, E)$, find *smallest* $V' \subseteq V$ s.t. if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ or both.

Decision problem is $\mathcal{NP}$-complete, optimisation problem is at least as hard.

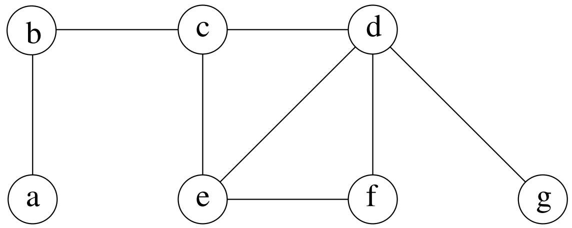
Trivial 2**-approximation** algorithm.

APPROX-VERTEX-COVER

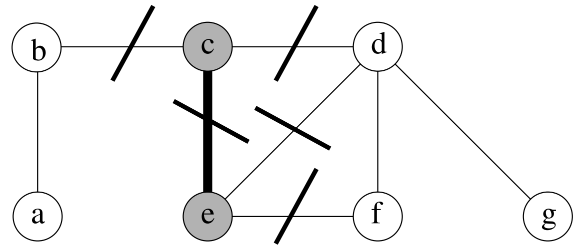
$C \leftarrow \emptyset$
$E' \leftarrow E$
while $E' \neq \emptyset$ **do**
    let (u, v) be an arbitrary edge of E'
    $C \leftarrow C \cup \{(u, v)\}$
    remove from E' all edges incident on either u or v
end while

Claim: after termination, C is a vertex cover of size at most twice the size of an optimal (smallest) one.

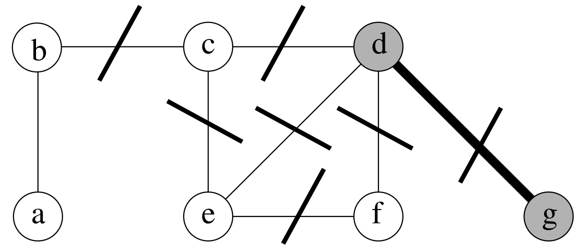
Example



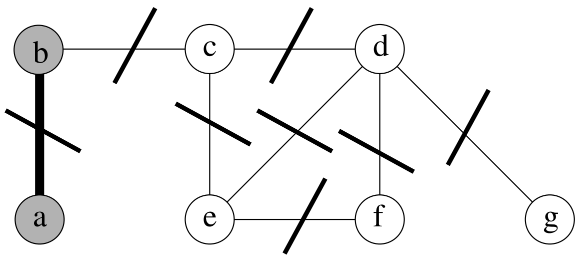
Input graph



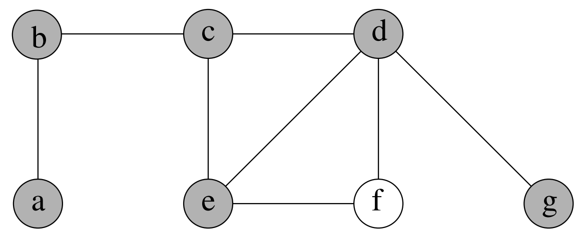
Step 1: choose edge (c,e)



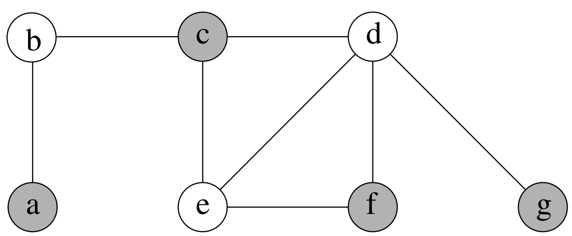
Step 2: choose edge (d,g)



Step 3: choose edge (a,b)



Result, size 6



Optimal result, size 4

Theorem. APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof. The **running time** is trivially bounded by $O(VE)$ (at most $|E|$ iterations, each of complexity at most $O(V)$). However, $O(V+E)$ can easily be shown.

Correctness: C clearly **is** a vertex cover.

Size of the cover: let A denote set of edges that are picked ($\{(c, e), (d, g), (a, b)\}$ in example).

- In order to cover edges in A, **any** vertex cover, in particular an **optimal** cover C^*, **must** include at least one endpoint of each edge in A.

- By construction of the algorithm, no two edges in A share an endpoint (once edge is picked, all edges incident on either endpoint are removed).

- Therefore, no two edges in A are covered by the same vertex in C^*, and

$$|C^*| \geq |A|.$$

- When an edge is picked, neither endpoint is already in C, thus

$$|C| = 2 \cdot |A|.$$

Combining (1) and (2) yields

$$|C| = 2 \cdot |A| \leq 2 \cdot |C^*|$$

(q.e.d.)

Interesting observation: we could prove that size of VC returned by alg is at most twice the size of optimal cover, **without knowing the latter**.

How? We **lower-bounded** size of optimal cover ($|C^*| \geq |A|$).

One can show that A is in fact a **maximal matching** in G.

- The size of any maximal matching is always a **lower bound** on the size of an optimal vertex cover (each edge has to be covered).
- The alg returns VC whose size is twice the size of the maximal matching A.

Example 2: The travelling-salesman problem

Problem: given complete, undirected graph $G = (V, E)$ with non-negative integer cost $c(u, v)$ for each edge, find cheapest hamiltonian cycle of G.

Consider two cases: with and without **triangle inequality**.

c satisfies triangle inequality, if it is always cheapest to go directly from some u to some w; going by way of intermediate vertices can't be less expensive.

Related decision problem is $\mathcal{NP}$-complete in both cases.

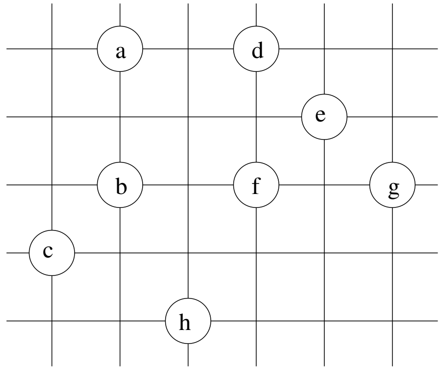
TSP with triangle inequality

We use function MST-PRIM(G, c, r), which computes an MST for G and weight function c, given some arbitrary root r.

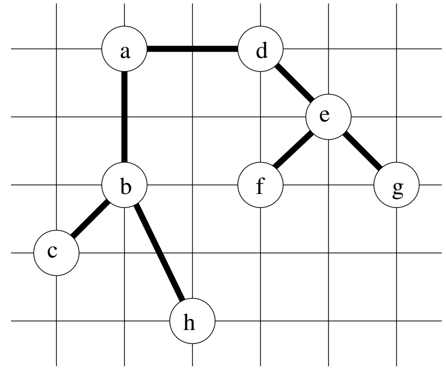
Input: $G = (V, E)$, $c : E \to \mathbf{R}$

APPROX-TSP-TOUR

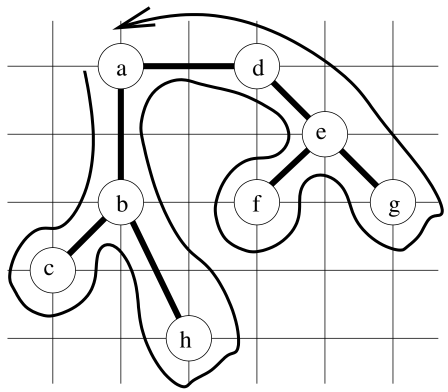
1: Select arbitrary $v \in V$ to be "root"
2: Compute MST T for G and c from root r using MST-PRIM(G, c, r)
3: Let L be list of vertices visited in pre-order tree walk of T
4: Return the hamiltonian cycle that vistis the vertices in the order L



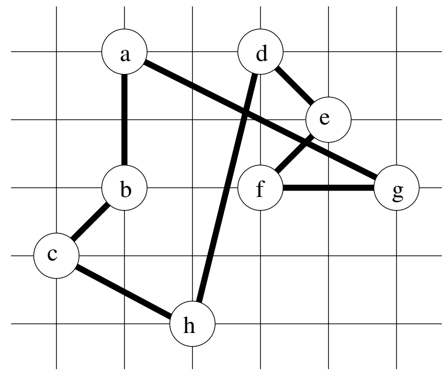
Set of points, lie in grid



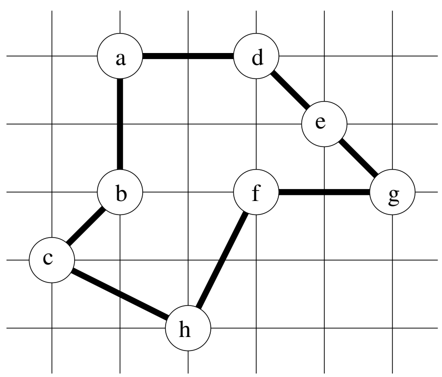
MST, root a



Pre−order walk



Resulting tour, cost ca. 19.1



Optimal tour, cost ca. 14.7

Theorem. APPROX-TSP-TOUR is a poly-time 2-approximation algorithm for the TSP problem with triangle inequality.

Proof.

Polynomial running time obvious, simple MST-PRIM takes $\Theta(V^2)$, computing preorder walk takes no longer.

Correctness obvious, preorder walk is always a tour.

Approximation ratio: Let H^* denote an optimal tour for given set of vertices.

Deleting any edge from H^* gives a spanning tree.

Thus, weight of **minimum** spanning tree is lower bound on cost of optimal tour:

$$c(T) \leq c(H^*)$$

A **full walk** of T lists vertices when they are **first visited**, and also when they are **returned to**, after visiting a subtree.

Ex: a,b,c,b,h,b,a,d,e,f,e,g,e,d,a

Full walk W traverses every edge **exactly twice** (although some vertex perhaps way more often), thus

$$c(W) = 2c(T)$$

Together with $c(T) \leq c(H^*)$, this gives $c(W) = 2c(T) \leq 2c(H^*)$

Problem: W is in general **not** a proper tour, since vertices may be visited more than once…

But: by our friend, the **triangle inequality**, we can **delete** a visit to any vertex from W and cost does **not increase**.

Deleting a vertex v from walk W between visits to u and w means going from u **directly** to w, without visiting v.

This way, we can consecutively remove all multiple visits to any vertex.

Ex: full walk a,b,c,b,h,b,a,d,e,f,e,g,e,d,a becomes a,b,c,h,d,e,f,g.

This ordering (with multiple visits deleted) is **identical** to that obtained by preorder walk of T (with each vertex visited only once).

It certainly is a Hamiltonian cycle. Let's call it H.

H is just what is computed by APPROX-TSP-TOUR.

H is obtained by deleting vertices from W, thus

$$c(H) \leq c(W)$$

Conclusion:

$$c(H) \leq c(W) \leq 2c(H^*)$$

(q.e.d.)

Although factor 2 looks nice, there are better algorithms.

There's a $3/2$ approximation algorithm by Christofedes (**with** triangle inequality).

Arora and Mitchell have shown that there is a PAS if the points are in the Euclidean plane (meaning the triangle inequality holds).

The general TSP

Now c does no longer satisfy triangle inequality.

Theorem. If $\mathcal{P} \neq \mathcal{NP}$, then for any constant $\rho \geq 1$, there is no poly-time ρ-approximation algorithm for the general TSP.

Proof. By contradiction. Suppose there **is** a poly-time ρ-approximation algorithm A, $\rho \geq 1$ integer. We use A to solve HAMILTON-CYCLE in poly time (this implies $\mathcal{P} = \mathcal{NP}$).

Let $G = (V, E)$ be instance of HAMILTON-CYCLE. Let $G' = (V, E')$ the **complete graph** on V:

$$E' = \{(u, v) : \ u, v \in V \wedge u \neq v\}$$

We assign **costs** to edges in E':

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ \rho \cdot |V| + 1 & \text{otherwise} \end{cases}$$

Creating G' and c from G certainly possible in poly time.

Consider TSP instance $\langle G', c\rangle$.

If original graph G has a Hamiltonian cycle H, then c assigns cost of one to reach edge of H, and G' contains tour of cost $|V|$.

Otherwise, any tour of G' **must** contain some edge **not** in E, thus have cost at least

$$\underbrace{(\rho \cdot |V| + 1)}_{\notin E} + \underbrace{(|V| - 1)}_{\in E} = \rho \cdot |V| + |V| \geq (\rho + 1) \cdot |V|$$

There is a **gap** of $\geq \rho \cdot |V|$ between cost of tour that is Hamiltonian cycle in G $(= |V|)$ and cost of any other tour.

Apply A to $\langle G', c\rangle$.

By assumption, A returns tour of cost at most ρ times the cost of optimal tour. Thus, if G contains Hamiltonian cycle, A **must** return it.

If G is not Hamiltonian, A returns tour of cost $> \rho \cdot |V|$.

We can use A to decide HAMILTON-CYCLE. *(q.e.d.)*

The proof was example of **general technique** for proving that a problem **cannot** be approximated well.

Suppose given $\mathcal{NP}$-hard problem X, produce minimisation problem Y s.t.

- "*yes*" instances of X correspond to instances of Y with value at most some k,
- "*no*" instances of X correspond to instances of Y with value greater than ρk

Then there is **no** ρ-approximation algorithm for Y unless $\mathcal{P} = \mathcal{NP}$.

Set-Covering Problem

Input: A finite set X and a family $\mathcal{F}$ of subsets over X. Every $x \in X$ belongs to at least one $F \in \mathcal{F}$.

Output: A minimum $S \subset \mathcal{F}$ such that

$$X = \bigcup_{F \in S} F.$$

We say that such S covers X and $x \in X$ is covered by $S' \subset \mathcal{F}$ if there exists a set $S_i \in S'$ that contains x.

The problem is a generalisation of the vertex cover problem.

It has many applications (cover a set of skills with workers,...)

We use a simple greedy algorithm to solve approximate the problem.

The idea is to add in every round a set S to the solution that covers the largest number of uncovered elements.

APPROX-SET-COVER

$U \leftarrow X$
$S \leftarrow \emptyset$
while $U \neq \emptyset$ **do**
    Select an $S_i \in \mathcal{F}$ that maximzes $|S_i \bigcap U|$
    $U \leftarrow U - S_i$
    $S \leftarrow S \cup S_i$
end while

The algorithm returns S.

Theorem. APPROX-SET-COVER is a poly-time $\log n$-approximation algorithm where $n = \{\max |F| : F \in \mathcal{F}\}$.

Proof. The running time is clearly polynomially in $|X|$ and $|\mathcal{F}|$.

Correctness: S clearly **is** a set cover.

Remains to show: S is a $\log n$ approximation

We will use **harmonic numbers**:

$$H(d) = \sum_{i=1}^{d} \frac{1}{d}.$$

$H(0) = 0$ and $H(d) = O(\log d)$.

Analysis

- Let S_i be the ith subset selected by APPROX-SET-COVER
- We assign a one to each set S_i selected by the algorithm.
- We will distribute the cost evenly over all elements that are covered for the first time.
- Let c_x be the cost assigned to $x \in X$. Then

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

- Let C be the cost of APPROX-SET-COVER. Then

$$C = \sum_{x \in X} c_x.$$

Analysis II

- Since each $x \in X$ is in at least one set $S' \in S^*$ we have

$$\sum_{S' \in S^*} \sum_{x \in S'} c_x \geq \sum_{x \in X} c_x := C$$

- Hence,

$$C \leq \sum_{S' \in S^*} \sum_{x \in S'} c_x.$$

Lemma. For any set $F \in \mathcal{F}$ we have

$$\sum_{x \in F} c_x \leq H(|F|).$$

Using the lemma we get

$$C \leq \sum_{S' \in S^*} \sum_{x \in S'} c_x \leq \sum_{S' \in S^*} H(S') \leq C^* \cdot H(\max\{|F| : F \in \mathcal{F}\}).$$

Lemma. For any set $F \in \mathcal{F}$ we have

$$\sum_{x \in F} c_x \leq H(|F|).$$

Proof. Consider any set $F \in \mathcal{F}$ and $i = 1, 2, \ldots C$ and let

$$u_i = |F - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|.$$

u_i is the number of elements in F that are not covered by $S_1, S_2, \ldots S_i$.

We also define $u_0 = |F|$.

Now let k be the smallest index such that $u_k = 0$.

Then $u_{i-1} \geq u_i$ and $u_{i-1} - u_i$ elements of F are covered for the first time by S_i (for $i = 1, \ldots k$).

We have

$$\sum_{x \in F} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$

Observe that

$$|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |F - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_i.$$

(the alg. chooses S_i such that the number of newly covered elements is max.).

Hence

$$\sum_{x \in F} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}$$

$$
\begin{aligned}
\sum_{x \in F} c_x &\leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\
&= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\
&\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \\
&= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\
&= \sum_{i=1}^{k} \left(H(u_{i-1}) - H(u_i) \right) \\
&= H(u_0) - H(u_k) = H(u_0) - H(0) \\
&= H(u_0) = H(|F|)
\end{aligned}
$$

Randomised approximation

A **randomised** algorithm has an approximation ratio of $\rho(n)$ if, for any input of size n, the **expected** cost C is within a factor of $\rho(n)$ of cost C^* of optimal solution.

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

So, just like with "standard" algorithm, except the approximation ratio is for the **expected** cost.

Consider 3-CNF-SAT, problem of deciding whether or not a given formula in 3CNF is satisfiable.

3-CNF-SAT is $\mathcal{NP}$-complete.

Q: What could be a related optimisation problem?

A: MAX-3-CNF

Even if some formula is perhaps not satisfiable, we might be interested in satisfying **as many clauses as possible**.

Assumption: each clause consists of exactly three distinct literals, and does not contain both a variable and its negation (so, we can not have $x \vee \overline{x} \vee y$ or $x \vee x \vee y$).

Randomised algorithm:

Independently, set each variable to 1 with probability $1/2$, and to 0 with probability $1/2$.

Theorem. Given an instance of MAX-3-CNF with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the described randomised algorithm is a randomised $8/7$-approximation algorithm.

Proof. Define **indicator variables** $Y_1, Y_2, \ldots, Y_m$ with

$$Y_i = \begin{cases} 1 & \text{clause } i \text{ is satisfied by the alg's assignment} \\ 0 & \text{otherwise} \end{cases}$$

This means $Y_i = 1$ if at least one of the three literals in clause i has been set to 1.

By assumption, settings of all three literals are independent.

A clause is **not** satisfied iff all three literals are set to 0, thus

$$\mathsf{P}[Y_i = 0] = \left(\frac{1}{2}\right)^3 = \frac{1}{8}$$

and therefore

$$\mathsf{P}[Y_i = 1] = 1 - \left(\frac{1}{2}\right)^3 = \frac{7}{8}$$

and

$$\mathsf{E}[Y_i] = 0 \cdot \mathsf{P}[Y_i = 0] + 1 \cdot \mathsf{P}[Y_i = 1] = \mathsf{P}[Y_i = 1] = \frac{7}{8}$$

Let Y be number of satisfied clauses, i.e. $Y = Y_1 + \cdots + Y_m$.

By **linearity of expectation**,

$$\mathsf{E}[Y] = \mathsf{E}\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} \mathsf{E}[Y_i] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7}{8} \cdot m$$

m is upper bound on number of satisfied clauses, thus approximation ratio is at most

$$\frac{m}{\frac{7}{8} \cdot m} = \frac{8}{7}$$

(q.e.d.)

An approximation scheme

An instance of the SUBSET-SUM problem is a pair $\langle S, t\rangle$ with $S = \{x_1, x_2, \ldots, x_n\}$ a set of positive integers, and t a positive integer.

The **decision problem** asks whether there is a subset of S that adds up to t.

SUBSET-SUM is $\mathcal{NP}$-complete.

In the **optimisation problem** we wish to find a subset of S whose sum is as large as possible but not larger than t.

An exponential-time algorithm

Just enumerate all subsets of S and pick the one with largest sum that does not exceed t.

There are 2^n possible subsets (an item is “in” or “out”), so this takes time $O(2^n)$.

Implementation could look as follows.

In iteration i, the alg computes sums of all subsets of $\{x_1, x_2, \ldots, x_i\}$.

As **starting point**, it uses all sums of subsets of $\{x_1, x_2, \ldots, x_{i-1}\}$.

Once a particular subset S' has sum **exceeding** t, no reason to maintain it: no superset of S' can possibly be a solution.

Iteratively compute L_i, list of sums of all subsets of $\{x_1, x_2, \ldots, x_i\}$ that do not exceed t.

Return the maximum value in L_n.

If L is a list of positive integers and x is another positive integer, then $L + x$ denotes list derived from L with each element of L **increased** by x.

Ex: $L = \langle 4, 3, 2, 4, 6, 7 \rangle$, $L + 3 = \langle 7, 6, 5, 7, 9, 10 \rangle$

We also use this notation for sets: $S + x = \{s + x : \ s \in S\}$.

Let MERGE-LIST(L, L') return sorted list that is merge of sorted L and L' with duplicates removed. Running time is $O(|L| + |L'|)$.

EXACT-SUBSET-SUM$(S = \{x_1, x_2, \ldots, x_n\}, t)$

$L_0 \leftarrow \langle 0 \rangle$
for $i \leftarrow 1$ **to** n **do**
    $L_i \leftarrow$ MERGE-LIST$(L_{i-1}, L_{i-1} + x_i)$
    remove from L_i every element that is greater than t
end for
return the largest element in L_n

How does it work?
Let P_i denote set of all values that can be obtained by selecting a (possibly empty) subset of $\{x_1, x_2, \ldots, x_i\}$ and summing its members.

Ex: $S = \{1, 4, 5\}$, then

$$
\begin{aligned}
P_1 &= \{0, 1\} \\
P_2 &= \{0, 1, 4, 5\} \\
P_3 &= \{0, 1, 4, 5, 6, 9, 10\}
\end{aligned}
$$

Clearly,

$$P_i = P_{i-1} \cup (P_{i-1} + x_i)$$

Can prove by induction on i that L_i is a sorted list containing every element of P_i with value at most t.

Length of L_i can be 2^i, thus EXACT-SUBSET-SUM is an exponential time algorithm **in general**.

However, in **special cases** it is poly-time if t is polynomial in $|S|$, or if all x_i are polynomial in $|S|$.

A fully-polynomial approximation scheme

Recall: running time must be polynomial in both $1/\epsilon$ and n.

Basic idea: modify exact exponential time algorithm by *trimming* each list L_i after creation:

If two values are "close", then we **don't maintain both of them** (will give similar approximations).

Precisely: given "trimming parameter" δ with $0 < \delta < 1$, then from a given list L we remove as many elements as possible, such that if L' is the result, for every element y that is removed, there is an element z still in L' that "aproximates" y:

$$\frac{y}{1+\delta} \leq z \leq y$$

Note: "one-sided error"

We say z **represents** y in L'.

Each removed y is represented by some z satisfying the condition from above.

Example:

$\delta = 0.1$, $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$

We can trim L to $L' = \langle 10, 12, 15, 20, 23, 29 \rangle$

  11 is represented by 10

  $21, 22$ are represented by 20

  24 is represented by 23

Given list $L = \langle y_1, y_2, \ldots, y_m \rangle$ with $y_1 \leq y_2 \leq \cdots \leq y_m$, the following function trims L in time $\Theta(m)$.

TRIM(L, δ)

$L' = \langle y_1 \rangle$
last$= y_1$
for $i \leftarrow 2$ **to** m **do**
  if $y_i >$ last $\cdot (1 + \delta)$ **then**
    / $y_i \geq$last because L is sorted */*
    append y_i onto end of L'
    last$\leftarrow y_i$
  end if
end for

Now we can construct our **approximation scheme**. Input is $S = \{x_1, x_2, \ldots, x_n\}$, x_i integer, target integer t, and "approximation parameter" ϵ with $0 < \epsilon < 1$.

It will return value z whose value is within $1 + \epsilon$ factor of optimal solution.

APPROX-SUBSET-SUM$(S = \{x_1, x_2, \ldots, x_n\}, t, \epsilon)$

```
L_0 ← ⟨0⟩
for i ← 1 to n do
    L_i ← MERGE-LIST(L_{i−1}, L_{i−1} + x_i)
    L_i ← TRIM(L_i, ε/2n)
    remove from L_i every element that is greater than t
end for
return z*, the largest element in L_n
```

Example

$S = \{104, 102, 201, 101\}$, $t = 308$, $\epsilon = 0.4$

$\delta = \epsilon/2n = 0.4/8 = 0.05$

line	
1	$L_0 = \langle 0 \rangle$
3	$L_1 = \langle 0, 104 \rangle$
4	$L_1 = \langle 0, 104 \rangle$
5	$L_1 = \langle 0, 104 \rangle$
3	$L_2 = \langle 0, 102, 104, 206 \rangle$
4	$L_2 = \langle 0, 102, 206 \rangle$
5	$L_2 = \langle 0, 102, 206 \rangle$
3	$L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$
4	$L_3 = \langle 0, 102, 201, 303, 407 \rangle$
5	$L_3 = \langle 0, 102, 201, 303 \rangle$
3	$L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$
4	$L_4 = \langle 0, 101, 201, 302, 404 \rangle$
5	$L_4 = \langle 0, 101, 201, 302 \rangle$

Alg returns $z^* = 302$, well within $\epsilon = 40\%$ of optimal answer $307 = 104 + 102 + 101$ (in fact, within 2%).

Theorem. APPROX-SUBSET-SUM is fully polynomial approximation scheme for the subset-sum problem.

Proof. Trimming L_i and removing from L_i every element that is greater than t maintain property that every element of L_i is member of P_i. Thus, z^* **is** sum of some subset of S.

Let $y^* \in P_n$ denote an optimal solution.

Clearly, $z^* \leq y^*$ (have removed elements that are too large).

Need to show $y^*/z^* \leq 1 + \epsilon$ **and** that running time is polynomial in n and $1/\epsilon$.

Can be shown (by induction) that $\forall y \in P_i$ with $y \leq t$ there is some $z \in L_n$ with

$$\frac{y}{(1 + \epsilon/2n)^i} \leq z \leq y$$

This also holds for $y^* \in P_n$, thus there is some $z \in L_n$ with

$$\frac{y^*}{(1+\epsilon/2n)^n} \leq z \leq y^*$$

and therefore

$$\frac{y^*}{z} \leq \left(1 + \frac{\epsilon}{2n}\right)^n$$

z^* is largest value in L_n, thus

$$\frac{y^*}{z^*} \leq \left(1 + \frac{\epsilon}{2n}\right)^n$$

Remains to show that $y^*/z^* \leq 1 + \epsilon$.

We know $(1 + a/n)^n \le e^a$, and therefore

$$\begin{aligned}\left(1+\frac{\epsilon}{2n}\right)^n &= \left(1+\frac{\epsilon}{2n}\right)^{2n\cdot(1/2)}\\ &= \left(\left(1+\frac{\epsilon}{2n}\right)^{2n}\right)^{1/2}\\ &= \left(\left(1+\frac{\epsilon}{2n}\right)^{2n}\right)^{1/2}\\ &\le (e^\epsilon)^{1/2}\\ &= e^{\epsilon/2}\end{aligned}$$

This, together with

$$e^{\epsilon/2} \le 1 + \epsilon/2 + (\epsilon/2)^2 \le 1 + \epsilon$$

gives

$$\frac{y^*}{z^*} \le \left(1+\frac{\epsilon}{2n}\right)^n \le 1+\epsilon$$

Approximation ratio OK, but what with **running time**?

We derive bound on $|L_i|$, Running time of APPROX-SUBSET-SUM is polynomial in lengths of L_i.

After trimming, successive elements z and z' of L_i fulfill $z'/z > 1 + \epsilon/2n$.

Thus, each list contains 0, possibly 1, and at most $\lfloor \log_{1+\epsilon/2n} t \rfloor$ additional values. We have

$$\begin{aligned}
|L_i| &\leq (\log_{1+\epsilon/2n} t) + 2 \\
&= \frac{\ln t}{\ln(1+\epsilon/2n)} + 2 \\
&\leq \frac{2n(1+\epsilon/2n)\ln t}{\epsilon} + 2 \\
&\quad \text{/* because of } x/(1+x) \leq \ln(1+x) \leq x \text{ */} \\
&\leq \frac{4n \ln t}{\epsilon} + 2 \\
&\quad \text{/* because of } 0 < \epsilon < 1 \text{ */}
\end{aligned}$$

This is polynomial in size of input ($\log t$ bits for t, plus bits for $x_1, x_2, \ldots, x_n$). Thus, it's polynomial in n and $1/\epsilon$.

Bin Packing

We are given n items with sizes $a_1, a_2, \ldots a_n$ with $a_i \in (0, 1]$.

The goal is to pack the items into m bins with capacity 1 each, and, thereby, to minimise the number of used bins.

Approximation is clear: find a value that is as close as possible to the optimal value for m.

Very easy: 2-approximation

This can be done using the *First Fit* algorithm:

- consider the items in an arbitrary order
- try to fit item into one of the existing bins, if not possible use a new bin for the item.

Easy to see that it calculates a two-approximation:

If the algorithm uses m bins then at least $m-1$ of them are more than half full. Therefore

$$a_1 + a_2 + \cdots + a_n \geq \frac{m-1}{2}.$$

Hence, $m - 1 < 2$ OPT and $m \leq 2$ OPT.

Theorem: For any $\epsilon > 0$, there is no bin packing algorithm having an approximation ratio of $3/2 - \epsilon$, unless $P = NP$.

Proof. Assume we have such an algorithm, than we can solve the SET PARTITIONING problem.

In SET PARTITIONING, we are given n non-negative numbers $a_1, a_2, \ldots, a_n$ and we would like to partition them into two sets having sum $(a_1 + a_2 + \cdots + a_n)/2$

This is the same than asking: can I pack the elements in two bins of size $(a_1 + a_2 + \cdots + a_n)/2$.

A $(3/2-\epsilon)$-approximation algorithm has to optput 2 for an instance of BIN BACKING that can be packed into two bins.

An asymptotic PTAS

Theorem: For any $0 < \epsilon \leq 1/2$, there is an algorithm A_ϵ that runs in time poly(n) and finds a packing using at most $(1 + 2\epsilon)$ OPT $+ 1$ bins.

The proof is split in two parts:

- It is easy to pack small items into bins. Hence, we consider the small items in the end.
- Only the big items have to be packed well.

Big Items

Lemma: Consider an instance I in which all n items have a size of at least ϵ. Then there is a poly(n) time $(1+\epsilon)$-approximation.

Proof.

- First we sort the items by increasing size.
- Then we partition the items into $K = \lceil 1/\epsilon^2 \rceil$ groups having at most $Q = \lfloor n\epsilon^2 \rfloor$ items. (Note: two groups can have items of the same size!)
- Construct instance J by rounding up the size of each item to the size of the largest item in the group.
- J has at most K different item sizes. Hence, there is a poly(n) time algorithm that solves J optimally:
  - The number of items per bin is bounded by $M = \lfloor 1/\epsilon \rfloor$.
  - The number of possible bin types is $R = \binom{M+K}{M}$ (which is constant).
  - Hence, the number of possible packings is at most $P = \binom{n+R}{R}$ (which is polynomial in n). We can enumerate all of them.

- Note: the packing we get is also valid for the original instance I
- To show

$$\mathsf{OPT}(J) \leq (1+\epsilon) \cdot \mathsf{OPT}(I).$$

  - Consider instance J' which is defined like J but we round down instead of rounding up. Clearly

$$\mathsf{OPT}(J') \leq \mathsf{OPT}(I).$$

  - Instance J' yields a packing for all items of J (and I) but the Q items of the largest group of J. Hence

$$\mathsf{OPT}(J) \leq \mathsf{OPT}(J') + Q \leq \mathsf{OPT}(I) + Q.$$

  - The largest group is packed into at most $Q = \lfloor n\epsilon^2 \rfloor$ bins.
  - We also have (min. item size is ϵ)

$$\mathsf{OPT}(I) \geq n\epsilon.$$

  - We have $Q = \lfloor n\epsilon^2 \rfloor \leq \epsilon\, \mathsf{OPT}$ and

$$\mathsf{OPT}(J) \leq (1+\epsilon) \cdot \mathsf{OPT}(I)$$

Small Items

They can be packed using first fit, the "hole" in every bin is at most ϵ.

APPROX-BIN-PACKING$(I = \{a_1, a_2, \ldots, a_n\})$

```
Remove items of size < ε
Round to optain constant number of item sizes
Find optimal Packing for the rounded items
Use this packing for original item sizes
Pack items of size < ε using First-Fit
```

Back to the Proof of the Theorem.

Let I be the input instance and I' the set of large items of I. Let M be the number of bins used by APPROX-BIN-PACKING.

We can find a packing for I' using at most $(1+\epsilon)\cdot\text{OPT}(I')$ many bins.

We pack the small items in First Fit manner into the bins opened for I' and open new bins if necessary.

- If no new bins are opened we have a $M \leq (1+\epsilon)\cdot\text{OPT}(I') \leq (1+\epsilon)\cdot\text{OPT}(I)$.
- If new bins are opened for the small items, all but the last bin are full to the extend of at least $1-\epsilon$.

  Hence the sum of item sizes in I is at least $(M-1)\cdot(1-\epsilon)$ and with $\epsilon \leq 1/2$

$$M \leq \frac{OPT}{1-\epsilon} + 1 \leq (1+2\epsilon)\cdot\text{OPT}(I) + 1.$$

The Knapsack Problem

Given: A set $S = \{a_1, a_2, \ldots a_n\}$ of objects with sizes $s_1, s_2, \ldots s_n \in Z^+$ and profits $p_1, p_2, \ldots p_n \in Z^+$ and a knapsack capacity B.

Goal: Find a subset of the objects whose total size is bounded by B and the total profit is maximised.

First Idea: Use a simple greedy algorithm that sorts the items by decreasing ratio of profit to size and pick objects in that order.

Homework: That algorithm can be arbitrarily bad!

Better:

APPROX-KNAPSACK($I = \{a_1, a_2, \ldots, a_n\}$)

1: Use the greedy algorithm to find a set of items S

2: Take the best of S and the item with largest profit

Theorem APPROX-KNAPSACK calculates a 2-approximation.

Proof.

Let k be the index of the first item that is not picked by the greedy algorithm.

Then $p_1 + p_2 + \cdots + p_k \geq OPT(I)$ (recall Problem Sheet 2)

Hence, either $p_1 + p_2 + \cdots + p_{k-1}$ or p_k is at least $\frac{\text{OPT}}{2}$.