FOUNDATIONS OF COMPUTER SCIENCE
LECTURE 17: Approximation algorithms

Prof. Daniele Gorla

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Handling NPC problems

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Many problems of practical significance are **NPC**, yet they are too important to abandon merely because we don't know how to find an optimal solution in polynomial time.

We have at least three ways to get around **NP**-completeness:

1. If the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
2. We may be able to isolate important special cases that we can solve in polynomial time.
3. We might come up with approaches to find *near-optimal* solutions in polynomial time (either in the worst case or the expected case).

   → In practice, near-optimality is often good enough.

We call an algorithm that returns near-optimal solutions an ***approximation algorithm***.

Approximation ratio

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution.

Depending on the problem, the optimal solution is one with maximum cost (maximization problem) or one with minimum cost (minimization problem).

Def.: We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

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

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a ***$\rho(n)$-approximation algorithm***.

For a maximization problem, $0 < C \le C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

For a minimization problem, $0 < C^* \le C$, and the ratio C/C^* gives the factor by which the cost of an optimal solution is smaller than the cost of the approximate solution.

The approximation ratio is never less than 1:

- a 1-approximation algorithm produces an optimal solution
- a large approximation ratio may return a solution that is much worse than optimal.

Approximation scheme

For many problems, we have polynomial-time approximation algorithms with approximation ratios that are small constants.

For other problems, the best known polynomial-time approximation algorithms have approximation ratios that grow as functions of the input size n.

Some NP-complete problems allow polynomial-time approximation algorithms that can achieve increasingly better approximation ratios by using more and more computation time.

→ we can trade computation time for the quality of the approximation.

<u>Def.:</u> An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\varepsilon > 0$ such that, for any fixed ε, the scheme is a $(1+ \varepsilon)$-approximation algorithm.

An approximation scheme is a ***polynomial-time approximation scheme (PTAS)*** if, for any fixed $\varepsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

An approximation scheme is a ***fully polynomial-time approximation scheme*** if its running time is polynomial in both $1/\varepsilon$ and the size n of the input instance.

Vertex Cover (1)

Recall that a ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset V' of V such that for every $\{u, v\} \in E$, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is $|V'|$.

The ***vertex-cover problem*** is to find a vertex cover of minimum size. We call such a vertex cover an ***optimal vertex cover***.

This problem is the optimization version of *VERTEX-COVER* ($\in$ **NPC**)

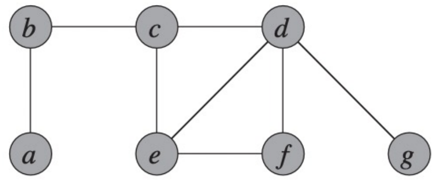
→ it can't be solved in polynomial-time, unless **P** = **NP**

IDEA: at every step, we select the node that «covers» more edges:

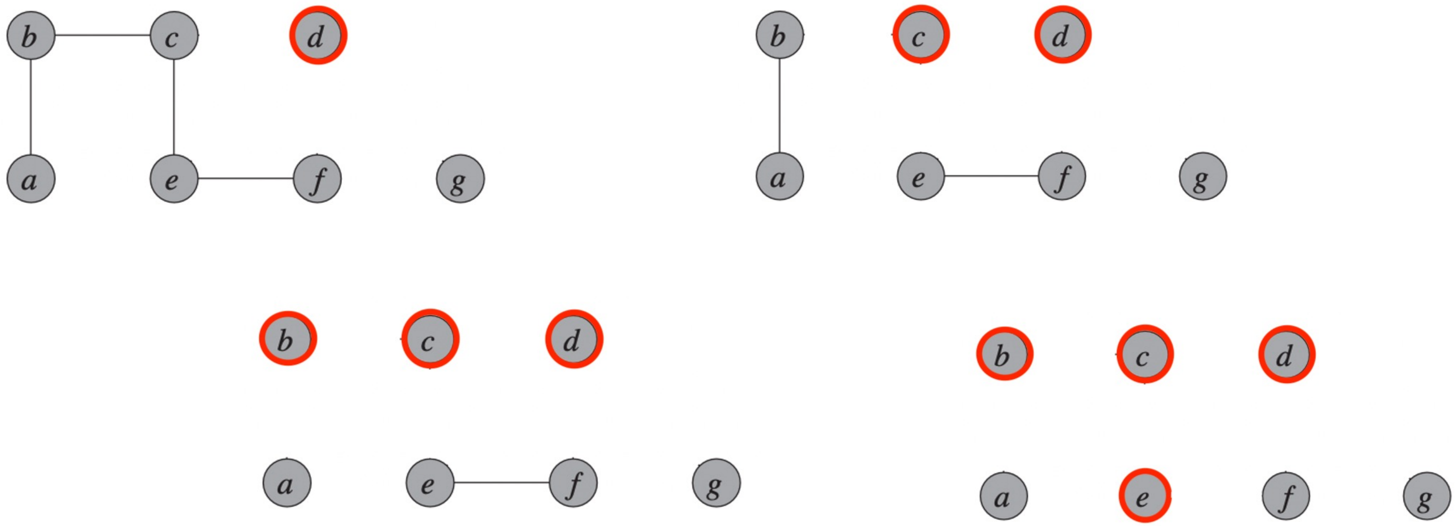
```
GREEDY-VERTEX-COVER(G)
    C ← ∅
    while E ≠ ∅
        Pick a vertex v ∈ V of
            maximum degree in the current graph
        C ← C ∪ {v}
        E ← E \ {e ∈ E : v ∈ e}
    return C
```

Vertex Cover (2)

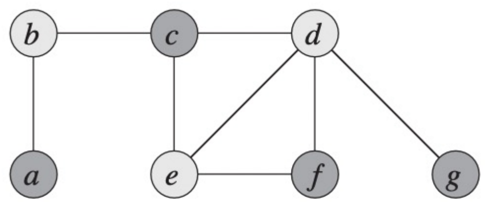
EXAMPLE: consider the graph



By running GREEDY-VERTEX-COVER we obtain in sequence:



whereas the optimal solution is:



Vertex Cover (3)

In this example, the ratio bewteen the returned solution and the optimal one is 4/3

→ but this should hold *for every* input graph!

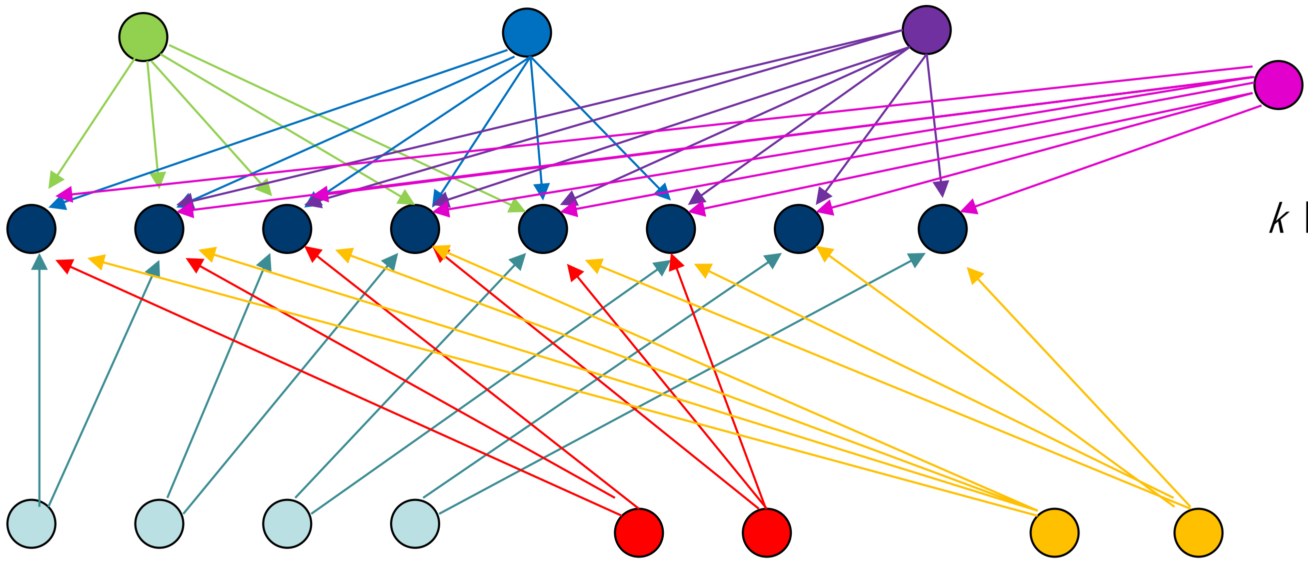
COUNTEREXAMPLE: consider the following graph:

$\left\lfloor \frac{k}{5} \right\rfloor$ nodes connected to disjoint 5-tuples of black nodes

$\left\lfloor \frac{k}{6} \right\rfloor$ nodes connected to disjoint 6-tuples of black nodes

$\left\lfloor \frac{k}{7} \right\rfloor$ nodes connected to disjoint 7-tuples of black nodes

$\left\lfloor \frac{k}{8} \right\rfloor$ nodes connected to disjoint 8-tuples of black nodes



k black nodes

$\left\lfloor \frac{k}{2} \right\rfloor$ nodes connected to disjoint pairs of black nodes

$\left\lfloor \frac{k}{3} \right\rfloor$ nodes connected to disjoint triples of black nodes

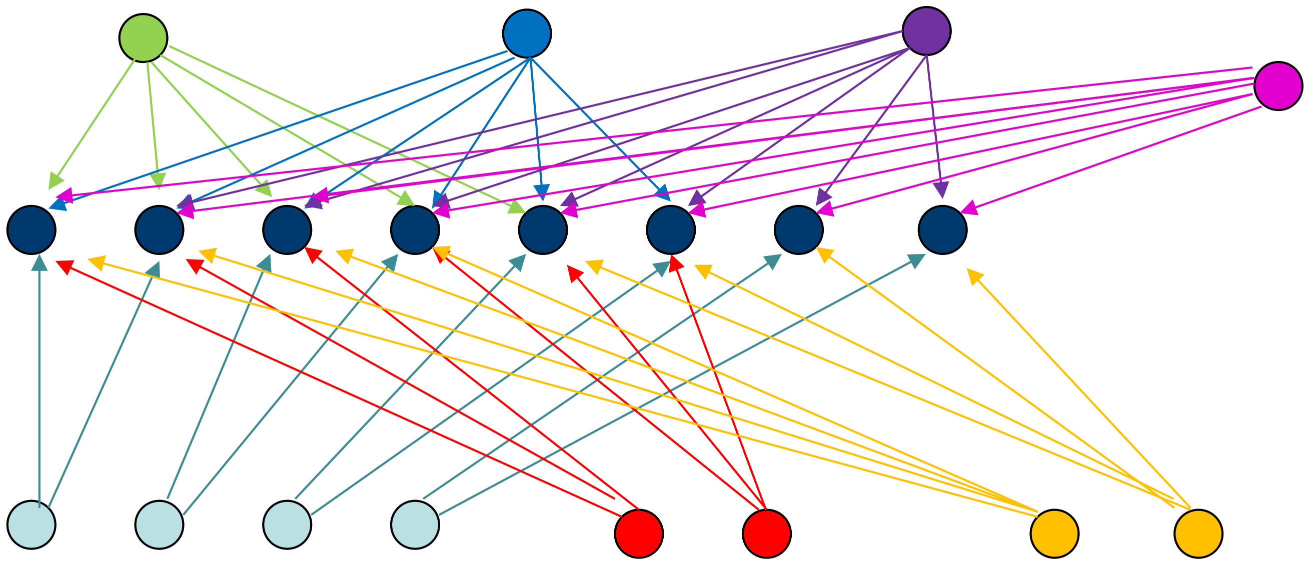
$\left\lfloor \frac{k}{4} \right\rfloor$ nodes connected to disjoint quadruples of black nodes

Vertex Cover (4)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

For applying GREEDY-VERTEX-COVER to this graph, the crucial observation is that there is always a coloured node with a degree higher than the degree of all black nodes:

1. We first select the pink node (deg.=8), since all other nodes have deg.at most 7
2. We then select the purple node (deg.=7), since all other nodes now have deg.at most 6
3. Then the blue node (deg.=6), since all other nodes now have deg. at most 5



4. Then the green node (deg.=5), since all other nodes now have deg. at most 4
5. Then the two orange nodes in any order (deg.=4), since all other nodes now have deg. at most 3
6. Then the red nodes (in any order)
7. Finally the light blue ones (in any order)

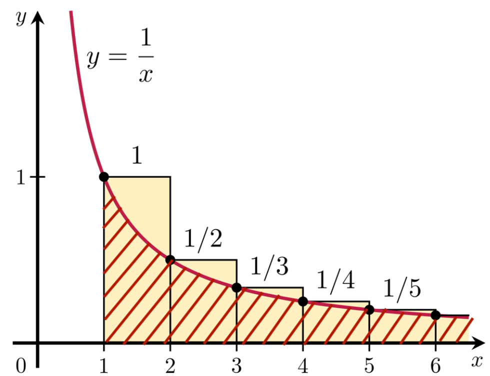
Vertex Cover (4)

GREEDY-VERTEX-COVER returns all the coloured nodes, that are

$$\sum_{i=2}^{k} \left\lfloor \frac{k}{i} \right\rfloor \geq \sum_{i=2}^{k} \left(\frac{k}{i} - 1 \right) = \sum_{i=2}^{k} \left(\frac{k}{i} - 1 \right) + (k-1) - (k-1) =$$

$$= \sum_{i=1}^{k} \left(\frac{k}{i} - 1 \right) - (k-1) = \sum_{i=1}^{k} \frac{k}{i} - k - (k-1) =$$

$$= k \sum_{i=1}^{k} \frac{1}{i} - 2k + 1 > k\left(\sum_{i=1}^{k} \frac{1}{i} - 2\right) > k\left(\int_{1}^{k} \frac{1}{x} dx - 2\right) = k\,(\ln k - 2)$$



The optimal vertex cover is formed by the black nodes (so, its cardinality is k).

So the approximation ratio in this case is at least $O(\log k)$ (where k is a function of n)

→ actually, it can be proved that this upper bound is tight, so this is the worst case

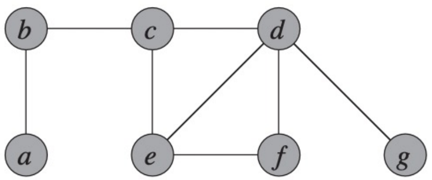
Vertex Cover (5)

Let us consider a different approach:

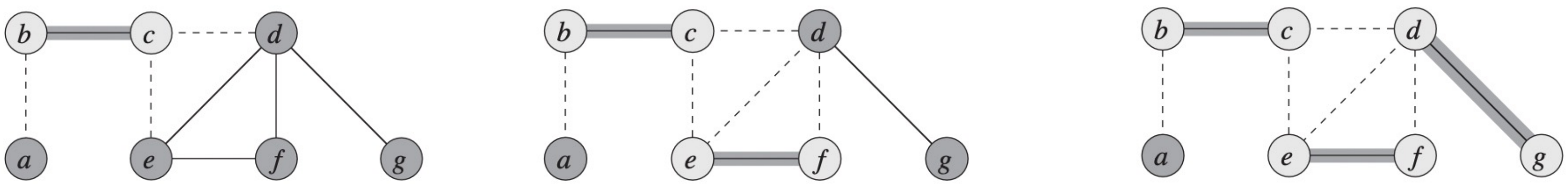
APPROX-VERTEX-COVER(G)

```
C = Ø
E' = G.E
while E' ≠ Ø
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```

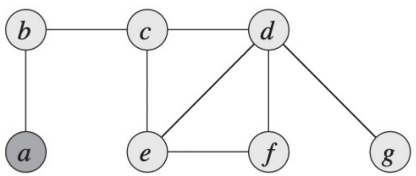
EXAMPLE:



Application of the algorithm:



Returned vertex cover:



Vertex Cover (6)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Thm.: APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof

APPROX-VERTEX-COVER runs in $O(|V| + |E|)$:

- Every node is added to C at most once, and
- Every edge is removed from E' at most once (either by selecting it or by canceling it)

The set C of vertices returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in E has been covered by some vertex in C.

APPROX-VERTEX-COVER returns a set whose size is at most twice the size of an optimal cover:

- Let A denote the set of edges picked in line 4.
- No two edges in A share an endpoint, so $|C| = 2|A|$
- Any vertex cover—in particular, an optimal cover C^*—must include at least one endpoint of each edge in A. Hence, $|C^*| \geq |A|$
- Thus, $|C| = 2|A| \leq 2|C^*|$. Q.E.D.

Traveling salesman (1)

In the ***traveling-salesman problem*** we have a complete undirected graph $G = (V, E)$ and a nonnegative integer cost c associated with each edge, and we must find a hamiltonian cycle of G with minimum cost.

→ this is the optimization problem associated to *TSP* (∈ **NPC**)

In many practical situations, the least costly way to go from u to v is to go directly, with no intermediate steps.

We formalize this notion by saying that the cost function c satisfies the ***triangle inequality*** if, for all vertices $u,v,w \in V$, we have that $c(u, v) \leq c(u, w) + c(w, v)$.

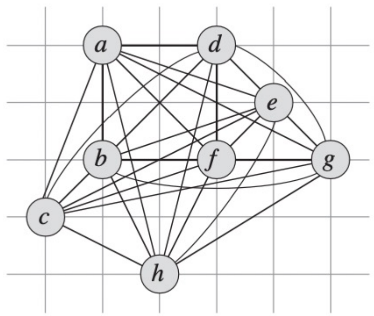
With the triangle inequality, we can provide the following approx.algorithm:

```
APPROX-TSP-TOUR(G, c)
select a vertex r ∈ G.V to be a "root" vertex
compute a minimum spanning tree T for G from root r
let H be a list of vertices, ordered according to when they are first visited
        in a preorder tree walk of T
return the hamiltonian cycle H
```

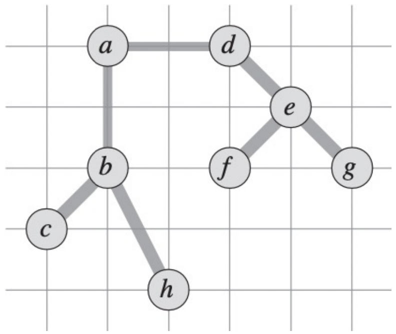
Traveling salesman (2)

EXAMPLE:

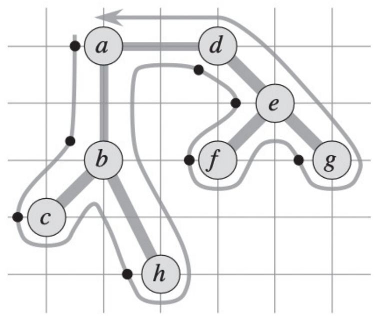
Input graph, with cost being the Euclidean distance:



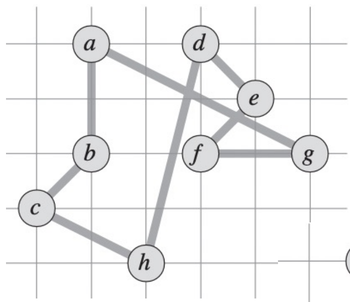
A minimum spanning tree with root *a* is:



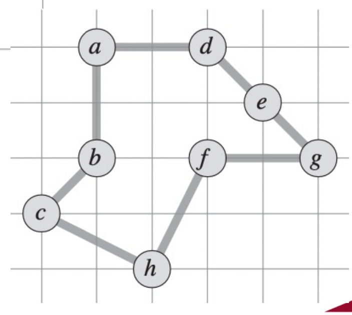
A preorder visit, that yields the vertex sequence *a, b, c, h, d, e, f, g*:



The approximate solution (with cost ~19,074):



The optimal solution (with cost ~14,715):



Traveling salesman (3)

Thm.: APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality.

Proof

The main cost of APPROX-TSP-TOUR is the minimum spanning tree; this can be computed in $O(|V|^2)$.

By def., any spanning tree includes all G's vertices and its preorder visit touches every vertex only once; so, the output of APPROX-TSP-TOUR is a hamiltonian cycle.

Let H^* be an optimal tour for the given graph.

Any spanning tree can be obtained by deleting any edge from a tour; since costs are nonnegative, we have that, if T is the minimum spanning tree calculated in line 2: $c(T) \leq c(H^*)$.

Let W be the sequence of vertices touched during a preorder visit of T.

Since W traverses every edge of T exactly twice, we have that

$c(W) = 2c(T) \leq 2c(H^*)$.

Traveling salesman (4)

Unfortunately, W is not hamiltonian, since it visits some vertices more than once.

By the triangle inequality, however, we can delete a visit to any vertex from W and the cost does not increase:

→ If $W = \ldots\ u \longrightarrow v \longrightarrow\ w\ \ldots$ and $W' = \ldots\ u \longrightarrow w\ \ldots$,

then $c(W') \leq c(W)$.

→ REMARK: this is always possible since G is a complete graph.

By repeatedly applying this operation, we can remove from W all but the first visit to each vertex, without increasing the cost.

The sequence of vertices obtained in this way is the same as that obtained by a preorder visit of T.

Let H be the cycle corresponding to this preorder visit. Hence,

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

<u>Q.E.D.</u>

Traveling salesman (5)

Thm.: If **P ≠ NP**, then for any constant $\rho \geq 1$, there is no poly-time approximation algorithm with approximation ratio ρ for the general traveling salesman problem.

Proof

By contradiction: assume a polynomial-time approximation algorithm *A* with approximation ratio $\rho \geq 1$.

W.l.o.g., we assume that ρ is an integer (by rounding it up if not).

We now show how to use *A* to solve instances of *U-HAM-CYCLE* in polynomial time.

Let $G = (V, E)$ be an instance of *U-HAM-CYCLE*; let's polynomially turn it into an instance of the traveling salesman by

1. considering as graph *G'* the complete graph on *V*; and
2. the cost function $c(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E \,, \\ \rho\,|V| + 1 & \text{otherwise} \,. \end{cases}$

Traveling salesman (6)

We now prove that G has a hamiltonian cycle IFF G' has a tour of cost $|V|$.

→ If G has a hamiltonian cycle H, then c assigns 1 to each edge of H, and so G' contains a tour of cost $|V|$.

← If G does not contain a hamiltonian cycle, then any tour of G' must use some edge not in E. Hence, any such tour has a cost of at least

$$(\rho|V| + 1) + (|V| - 1) = \rho|V| + |V| = (\rho + 1)|V| \quad (> |V|)$$

Thanks to this, we can use A to say in poly-time whether a G belongs to *U-HAM-CYCLE* or not:

1. Build G' and c as describe before
2. Run A on (G', c)
3. Because A is guaranteed to return a tour of cost no more than ρ times the cost of an optimal tour:
   - If A returns a tour of cost $\leq \rho|V|$, then $G \in$ *U-HAM-CYCLE*;
   - Otherwise, $G \notin$ *U-HAM-CYCLE.*

Q.E.D.

Subset sum (1)

The optimization version of this problem is to find a subset of a given set whose sum is as large as possible, but not larger than a given t.

For example:

- we have a truck that can carry no more than t pounds, and
- n different boxes to ship, the i-th of which weighs x_i pounds.
- We wish to fill the truck with as heavy a load as possible without exceeding the given weight limit.

The procedure EXACT-SUBSET-SUM:

- takes in input a set $S = \{x_1, x_2, \ldots, x_n\}$ and a target value t;
- iteratively computes L_i, the list of sums of all subsets of $\{x_1, \ldots, x_i\}$ that do not exceed t; and
- returns the maximum value in L_n.

This algorithm provides an exact solution of the problem, but it runs in exponential time:

→ the cardinality of L_i is (at most) the numb. of subsets of a set with i elements, that is 2^i

Subset sum (2)

```
EXACT-SUBSET-SUM(S, t)
n = |S|
L_0 = ⟨0⟩
for i = 1 to n
      L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
      remove from L_i every element that is greater than t
return the largest element in L_n
```

where:

- $L + x$ denotes the list L where each element is increased by x;
- MERGE-LISTS(L, L') takes two sorted lists and returns the sorted list that is the merge of its inputs, with duplicate values removed.

  → it runs in $O(|L|+|L'|)$

EXAMPLE: if $S = \{1, 4, 5\}$ and $t = 7$, then

- $L_0 = \langle 0 \rangle$
- L_1 = MERGE-LISTS($\langle 0 \rangle$, $\langle 1 \rangle$) = $\langle 0, 1 \rangle$
- L_2 = MERGE-LISTS($\langle 0, 1 \rangle$, $\langle 4, 5 \rangle$) = $\langle 0, 1, 4, 5 \rangle$
- L_3 = MERGE-LISTS($\langle 0, 1, 4, 5 \rangle$, $\langle 5, 6, 9, 10 \rangle$) = $\langle 0, 1, 4, 5, 6, 9, 10 \rangle$
- Remove from L_3 the last two numbers (that exceed 7)
- Return 6 (that is the sum of 1 and 5)

Subset sum (3)

Since we aim at an approximate solution, we can refine (or *trim*) the L_i s by removing values from a list if they are «close enough» to their preceeding value.

More precisely:

- We use a trimming parameter $\delta \in (0, 1)$
- We ***trim*** a list L according to δ by removing as many elements from L as possible, in such a way that, if L' is the result, then for every element y removed from L there is an element z in L' that approximates y:

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

Given a list $L = \langle y_1, y_2, \ldots, y_m \rangle$ sorted in monotonically increasing order and a δ, the procedure for trimming it is:

```
TRIM(L, δ)
let m be the length of L
L′ = ⟨y_1⟩
last = y_1
for i = 2 to m
    if y_i/(1 + δ) ≰ last          // last ≤ y_i because L is sorted
        append y_i onto the end of L′
        last = y_i
return L′
```

Subset sum (4)

EXAMPLE: Let $\delta = 0.1$ and $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$.

We can trim L as follows:

- 10 is inserted in L' by default
- 11 is represented by 10, since $11/1.1 = 10 \leq 10$;
- 12 is kept, since $12/1.1 = 10{,}90\ldots \not\leq 10$;
- 15 is kept, since $15/1.1 = 13{,}63\ldots \not\leq 12$;
- 20 is kept, since $20/1.1 = 18{,}18\ldots \not\leq 15$;
- 21 and 22 are represented by 20, since $21/1.1 = 19.09\ldots \leq 20$ and $22/1.1 = 20 \leq 20$;
- 23 is kept, since $23/1.1 = 20{,}90\ldots \not\leq 20$;
- 24 is represented by 23, since $24/1.1 = 21.81\ldots \leq 23$;
- 29 is kept, since $29/1.1 = 26{,}36\ldots \not\leq 23$.

Hence, the trimmed list is $L' = \langle 10, 12, 15, 20, 23, 29 \rangle$.

Subset sum (5)

The approximation algorithm we devise takes in input:

- a (non-sorted) set $S = \{x_1, \ldots, x_n\}$,
- a value t, and
- an $\varepsilon \in (0, 1)$ → here, we aim at a fully poly-time approx. scheme (PTAS); hence, the devised algorithm takes in input also how much we want to stay far from an optimal solution and it should run in poly-time also in $1/\varepsilon$

APPROX-SUBSET-SUM(S, t, ϵ)

$n = |S|$
$L_0 = \langle 0 \rangle$
for $i = 1$ **to** n
    $L_i = $ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
    $L_i = $ TRIM$(L_i, \epsilon/2n)$ // the trimming parameter δ is $\varepsilon/2n$
    remove from L_i every element that is greater than t
let z^* be the largest value in L_n
return z^*

REMARK: the only difference w.r.t. EXACT-APPROX-SUM is the trimming!

Subset sum (6)

EXAMPLE: Let $S = \{104, 102, 201, 101\}$, $t = 308$ and $\varepsilon = 0.4$.

→ The trimming parameter $\delta = \varepsilon/2n = \varepsilon/8 = 0.05$.

APPROX- SUBSET-SUM computes the following values:

$L_0 = \langle 0 \rangle$,

$L_1 = \langle 0, 104 \rangle$, no trimming, no cancellation of elements bigger than t.

$L_2 = \langle 0, 102, 104, 206 \rangle$,

we can get rid of 104, since 104/1.05 = 99,… < 102; so, $L_2 = \langle 0, 102, 206 \rangle$

$L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$

we can get rid of 206, since 206/1.05 = 196,… < 201; so,

$L_3 = \langle 0, 102, 201, 303, 407 \rangle$

we can cancel 407, being bigger than t; so, $L_3 = \langle 0, 102, 201, 303 \rangle$

$L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$

we can get rid of 102 and 203, since 102/1.05 = 97,… < 101 and

203/1.05 = 193,… < 201; so, $L_4 = \langle 0, 101, 201, 302, 404 \rangle$

we can cancel 404, being bigger than t; so, $L_4 = \langle 0, 101, 201, 302 \rangle$

The returned value is 302, whereas the optimal solution is 104+102+101=307.

→ the ratio between the opt.val. and the approx.val. is ~1,017 < 1.4 = 1+ε

Subset sum (7)

Thm.: APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.

Proof

All elements of every L_i are the sum of some subset of S by construction.

Now, for every $y \leq t$ that is the sum of a subset of $\{x_1, \ldots, x_i\}$, we have that there exists a $z \in L_i$ such that

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

Indeed, by definition of trimming, every y is «represented» by some z such that

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

Being $\varepsilon/2n > 0$ (since $\varepsilon > 0$), we have that $(1+ \varepsilon/2n)^i \geq (1+ \varepsilon/2n)$.

Let y^* denote an optimal solution to the subset-sum problem (so, it is the sum of a subset of S and $y^* \leq t$). Hence, there exists $z \in L_n$ s.t. $\frac{y^*}{(1 + \epsilon/2n)^n} \leq z \leq y^*$

Let z^* be the biggest element in L_n; hence, $z \leq z^*$ and so $\frac{y^*}{z^*} \leq \left(1 + \frac{\epsilon}{2n}\right)^n \leq 1 + \epsilon$ (by using some good old analysis).

Subset sum (8)

We have to bound the length of L_i, since each iteration of the for-loop has a complexity that is linear in $|L_i|$:

- After trimming, successive elements z_j and z_{j+1} of L_i must be s.t. $z_{j+1}/z_j > (1 + \varepsilon/2n)$ (otherwise z_{j+1} would have been represented by z_j)
- Therefore, each list contains the value 0, possibly the value 1, and it is of the form

  $\langle 0 , 1 , z_1 , z_2 , \dots z_k \rangle$ where $z_1 > (1+ \varepsilon/2n)$, $z_2 > z_1(1+ \varepsilon/2n) > (1+ \varepsilon/2n)^2$,
  $\dots z_k > z_{k-1}(1+ \varepsilon/2n) > (1+ \varepsilon/2n)^k$

  with $z_k \leq t$. Hence, $(1+ \varepsilon/2n)^k < t$, that is $k < \log_{1+\varepsilon/2n} t$.

- So, $|L_i| \leq \log_{1+\epsilon/2n} t + 2$

$$\begin{aligned} \log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\epsilon/2n)} + 2 \\ &\leq \frac{2n(1+\epsilon/2n)\ln t}{\epsilon} + 2 \\ &< \frac{3n\ln t}{\epsilon} + 2 \end{aligned}$$

where

- the first inequality arises from $\dfrac{x}{1+x} \leq \ln(1+x)$
- the second inequality from $\varepsilon < 1$ and $n \geq 1$.

Overall, APPROX-SUBSET-SUM runs in $O(\frac{n^2 \ln t}{\epsilon})$. Q.E.D.