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.

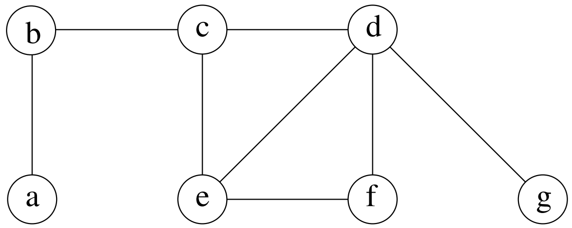
Trivial 2**-approximation** algorithm.

APPROX-VERTEX-COVER

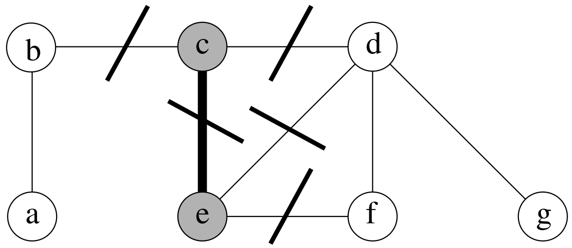
$C \leftarrow \emptyset$
$E' \leftarrow E$
while $E' \neq \emptyset$ **do**
    let (u, v) be an arbitrary edge of E'
    $C \leftarrow C \cup \{(u, v)\}$
    remove from E' all edges incident on either u or v
end while

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

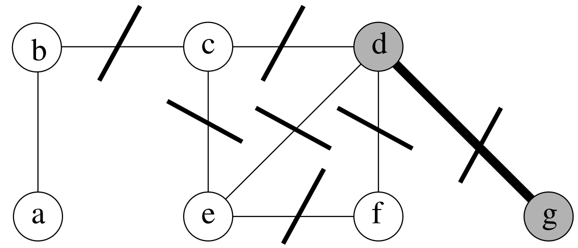
Example



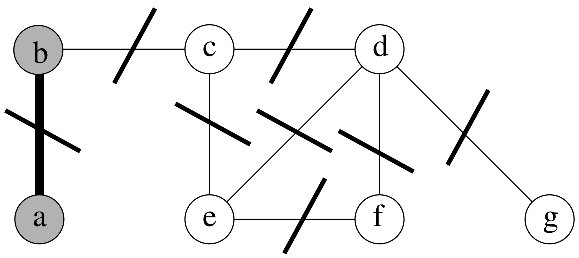
Input graph



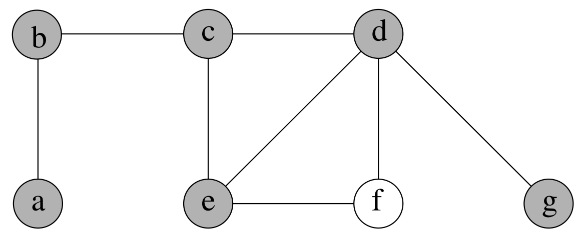
Step 1: choose edge (c,e)



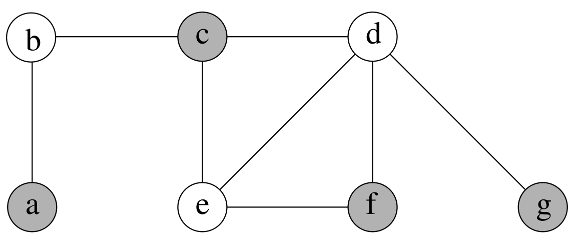
Step 2: choose edge (d,g)



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.

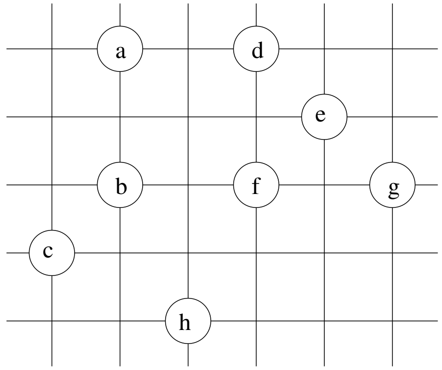
TSP with triangle inequality

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

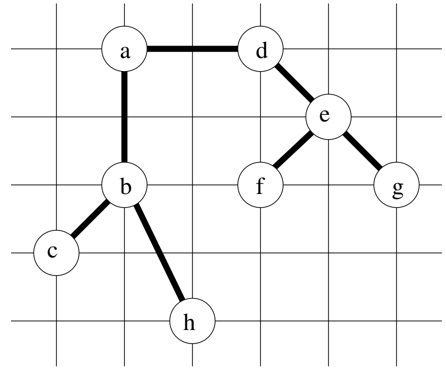
Input: $G = (V, E)$, $c : E \to \mathbf{R}$

APPROX-TSP-TOUR

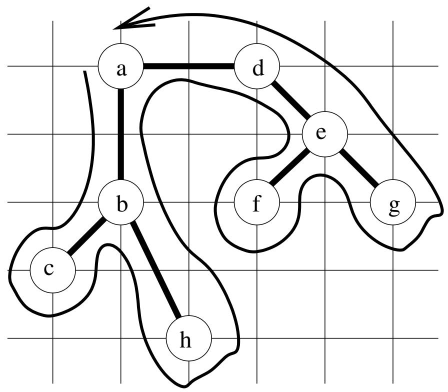
1: Select arbitrary $v \in V$ to be "root"
2: Compute MST T for G and c from root r using MST-PRIM(G, c, r)
3: Let L be list of vertices visited in pre-order tree walk of T
4: Return the hamiltonian cycle that vistis the vertices in the order L



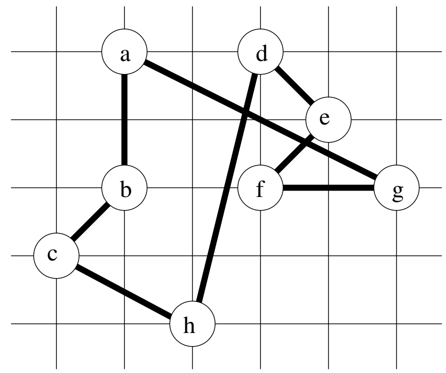
Set of points, lie in grid



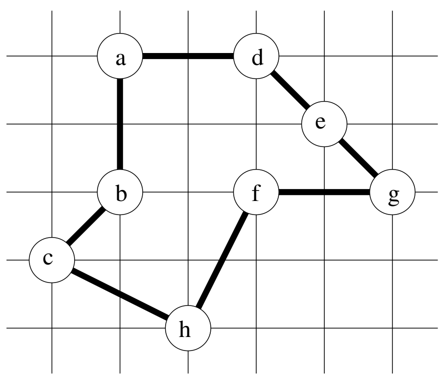
MST, root a



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 $c(T) \le c(H^*)$, this gives $c(W) = 2c(T) \le 2c(H^*)$

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 ← X
S ← ∅
while U ≠ ∅ do
    Select an S_i ∈ 𝓕 that maximzes |S_i ⋂ U|
    U ← U − S_i
    S ← S ∪ 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

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

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