Approximation algorithms

An optimization problem \( P \) defines:
1. a set of valid outputs (solutions)
2. the cost \( c(x) \) of an output \( x \)
3. whether to minimize or maximize cost

It is called a minimization or maximization problem.

**VERTEX COVER:**

- input: graph \( G \)
- output: vertex cover \( S \)
- objective: minimize \(|S|\)

Optimal algorithm computes best output. \( \text{OPT:} \)
\( c(\text{OPT}) = \min/\max \) cost among all valid outputs \( x \)

Approximation algorithm can compute any valid output

\( \alpha \)-approximation algorithm computes an output \( \text{APX} \)
whose cost is within an \( \alpha \geq 1 \) factor of \( \text{OPT}: \)
\[
\frac{1}{\alpha} \cdot c(\text{OPT}) \leq c(\text{APX}) \leq \alpha \cdot c(\text{OPT})
\]

---

\*for maximization

\*for minimization

\( \alpha \) is the approximation ratio ~allow dependence on \( n \)
2-approximation algorithm for VERTEX COVER:

1. pick any edge \((u,v) \in E\)
2. add both \(u\) & \(v\) to \(S\)
3. delete \(u\) & \(v\) & incident edges from \(G\)
4. repeat until all edges are gone (covered)

- \(O(V+E)\) time
- output depends on order we visit edges
- output always valid solution: cover all edges

Claim: \(|APX| \leq 2 \cdot |OPT|\).

Proof: Compare \(APX\) vs. \(OPT\).

For every \((u,v)\) pair we add to \(APX\), we know either \(u \in OPT\) or \(v \in OPT\).

By induction, \(|APX - \{u,v\}| \leq 2 \cdot |OPT - \{u,v\}|\)

(Base case: \(G\) empty \(\Rightarrow |APX| = |OPT| = 0\.)

\[ |APX| = |APX - \{u,v\}| + 2 \]
\[ \leq 2 \cdot |OPT - \{u,v\}| + 2 \]
\[ \leq 2 \cdot (|OPT| - 1) + 2 \]
\[ = 2 \cdot |OPT|. \]

Best approximation algorithm: \(2 - \frac{1}{2} \log_2 \sqrt{V}\).

Lower bound: no polynomial-time \((1 + \frac{1}{6})\)-approximation unless \(P = NP\).
Traveling Salesman Problem (TSP)

input: complete graph \( K_m \), edge-weight function \( w \)
output: cycle \( v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_m \rightarrow v_1 \)
visiting each vertex exactly once
objective: minimize weight of cycle:
\[
w(v_1, v_2) + w(v_2, v_3) + \ldots + w(v_{m-1}, v_m) + w(v_m, v_1)
\]

- famous NP-complete problem

Hard special case: each \( w(u, v) \in \{0, \infty\} \).
\[\Rightarrow\] every cycle has weight 0 if all edges do
& \( \infty \) otherwise
\[\Rightarrow\] \( \alpha \)-approximation algorithm, for any \( \alpha < \infty \),
must in fact be optimal
i.e. correctly determine whether there is
a cycle visiting each vertex exactly once
& using only weight-0 edges
- define graph \( G = (V, E) \), \( |V| = m \),
where \( (u, v) \in E \iff w(u, v) = 0 \)
\[\Rightarrow\] \( \alpha \)-approximation must determine whether
\( G \) has a cycle visiting each vertex exactly once
Hamiltonian cycle
- this problem is NP-complete
\[\Rightarrow\] no polynomial-time \( \alpha \)-approximation unless \( \text{P=NP} \)
[in reality, \( \infty \) replaced by \( n^c \) or \( c^n \)]
TSP with triangle inequality: \( w(u,v) \leq w(u,x) + w(x,v) \) for all \( u,v,x \).

E.g. \( w(u,v) = \) Euclidean distance between points \( u \) and \( v \).
\( w(u,v) = \) shortest-path weight \( u \rightarrow v \) in graph \( G \).

2-approximation algorithm:
1. compute minimum spanning tree \( T \)
2. pick arbitrary root vertex for \( T \)
3. output preorder traversal of \( T \).

Equivalently:
- walk "around" the tree
- visiting every tree edge exactly twice
- but skip over repeats of vertices
- take "shortcut" guaranteed by \( \Delta \) inequality direct to next new vertex

\[ \Rightarrow w(\text{ARX}) \leq 2 \cdot w(\text{MST}) \quad \text{by } \Delta \text{ inequality} \]
Approximation factor:

Claim: \( w(\text{OPT}) \geq w(\text{MST}) \)

Proof: Take OPT TSP cycle.
    Remove an edge.
    Resulting path \( P \) is a spanning tree
    with \( w(P) \leq w(\text{OPT}) \).
    \( w(\text{MST}) \leq w(P) \).

\[ \Rightarrow w(\text{APX}) \leq 2 \cdot w(\text{MST}) \leq 2 \cdot w(\text{OPT}) \]

Best approximation algorithm: 1.5 [Christofides]

Lower bound: \( < 1 + \frac{1}{3812} \) impossible unless \( P=NP \).
    (but 1.5 conjectured optimal)
An approximation scheme is an approximation algorithm with an additional input $\epsilon > 0$. The output is like a $(1+\epsilon)$-approximation.

$PTAS = \text{polynomial-time approximation scheme}$

$\Rightarrow \text{polynomial time in } n \text{ for any fixed } \epsilon > 0$

$\Rightarrow O(n f(\epsilon)) \text{ is O.K.}$

$EPTAS = \text{efficient PTAS}$

$\leq O(n^c) \text{ for any fixed } \epsilon > 0, c \geq 1 \text{ indep. of } \epsilon$

i.e. $O(f(\epsilon) \cdot n^c)$

$FPTAS = \text{fully PTAS}$

$\Rightarrow \text{polynomial time in } n \text{ & } \frac{1}{\epsilon}$

i.e. $O((n^c)^c)$.

Theorem: if problem $\Pi$ has an EPTAS then it is FPT with respect to $\frac{1}{\epsilon}$. 
**KNAPSACK:** ("Subset-Sum" in CLRS)

**input:** positive integers \(w_1, w_2, \ldots, w_n\) (weights) & positive integer \(t\) (sack capacity)

**output:** index set \(I \subseteq \{1, 2, \ldots, n\}\) (which items) such that \(\sum_{i \in I} w_i \leq t\)

**objective:** maximize \(\sum_{i \in I} w_i\)

**Dynamic programming algorithm:**

- subproblem = find all valid solutions for prefix \(w_1, w_2, \ldots, w_k\)

- solutions for \(w_1, w_2, \ldots, w_k\) = \{ \overline{I} \cup \{w_k\} \mid \text{I solution for } w_1, w_2, \ldots, w_{k-1} \}

- potentially doubles for each \(k\)

\(\Rightarrow O(2^n \cdot n)\) time.
FPTAS for KnapSack:

Idea: round each solution cost up to next power of $1 + \varepsilon$.

$\Rightarrow$ only $\log_{1+\varepsilon} t$ possible solution costs to keep track of.

Each step of $k$ introduces error of $(1+\varepsilon)^k x$.
Total error is $(1+\varepsilon)^n x$.
We keep solutions whose cost $\times$ error $\leq t$.
$\Rightarrow c(\text{APX}) \times \text{error} \geq c(\text{OPT})$
$\Rightarrow$ approximation ratio $\leq \text{error} \leq (1+\varepsilon)^n$.

Set $S = \varepsilon / 2n$.
$\Rightarrow (1+\varepsilon)^n = (1 + (\varepsilon / 2)/n)^n$
$\leq e^{\varepsilon/2} \leq 1 + \varepsilon/2 + (\varepsilon/2)^2$
$\leq 1 + \varepsilon$.

Time $= O(n \log_{1+\varepsilon} t)$

$= O(n \log t / \log (1 + \varepsilon / 2n))$

$\geq \frac{\varepsilon}{2n} - \frac{1}{2} (\varepsilon / 2n)^2$ Taylor
$\geq \varepsilon / 4n$

$= O(n^2 (\log t) / \varepsilon)$

$\Rightarrow$ FPTAS.

#bits to represent $t$
Follow-on classes:

6.854: Advanced Algorithms (falls)
6.856: Randomized Algorithms (odd springs)
6.850: Geometric Computing (odd springs)
6.851: Advanced Data Structures (odd springs)
6.852: Distributed Algorithms (even springs)
6.255: Optimization Methods (springs)
6.855: Network Optimization (springs)
6.047: Computational Biology (springs)
6.857: Network & Computer Security (falls)
6.976: Network Algorithms (this spring)
6.896: Sublinear Time Algorithms (this spring)

6.045: Automata, Computability, Complexity (springs)
6.840: Theory of Computation (falls)
6.841: Advanced Complexity Theory (springs)