10 Approximation Algorithms

8 July 2025

Prof. Dr. Sebastian Wild

CS627 (Summer 2025)
Philipps-Universität Marburg
version `2025-07-08 00:03` H

Outline

10 Approximation Algorithms

10.1 Motivation and Definitions

Recap: Optimization Problems, NPO

Recall general optimization problem $U \in \mathsf{NPO}$:

- each instance x has non-empty set of *feasible solutions* $M(x)$
- objective function $cost$ assigns value $cost(y)$ to all candidate solutions $y \in M(x)$
- can check in polytime
  - whether x is a valid instance
  - whether $y \in M(x)$
  - compute $cost(y) \in \mathbb{Q}$

For each U, consider two variants:

- *optimization problem:* output $y \in M(x)$ s.t. $cost(y) = \overset{\text{min or max}}{goal}_{y' \in M(x)} cost(y')$
- *evaluation problem:* output $goal_{y \in M(x)} cost(y)$

Perfect is the enemy of good

Optimal solutions are great, but if they are too expensive to get,
maybe ***"close-to-optimal"*** suffices?

A "consistent" with problem

A ***heuristic*** is an algorithm A that always computes a feasible solution $A(x) \in M(x)$,
but we may not have any guarantees about $cost(A(x))$.

(Sometimes that's all we have . . .)

Our goal: Prove guarantees about worst possible $cost(A(x))$.
Problem: optimal objective function value depends on x,
so how to define *"good enough"*?

Relate $cost(A(x))$ to ***OPT*** $= goal_{y \in M(x)} cost(y)$. $\rightsquigarrow$ *approximation algorithm*

Approximation Algorithms

Definition 10.1 (Approximation Ratio)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ be an optimization problem. For every $x \in L_I$ we denote its *optimal objective value* by $OPT = OPT_U(x) = goal_{y \in M(x)} cost(y)$.

Let further A be an algorithm consistent with U.

The ***approximation ratio $R_A(x)$ of A on x*** is defined as $R_A(x) = \dfrac{cost(A(x))}{OPT_U(x)}$.

Note: For minimization problems, $R_A \geq 1$; for maximization problems $R_A \leq 1$

Definition 10.2 (Approximation Algorithm)

An algorithm A consistent with an optimization problem $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ is called a ***c-approximation (algorithm) for U*** if

- $goal = \min$ and $\forall x \in L_I : R_A(x) \leq c$;
- $goal = \max$ and $\forall x \in L_I : R_A(x) \geq c$.

10.2 Vertex Cover and Matchings

Example: Vertex Cover

Recall the VertexCover optimization problem.
C is a VC iff $\{u, v\} \in E : \{u, v\} \cap C \neq \emptyset$
$goal = \min$
How can we vouch for a VC C to be (close to) optimal?

Definition 10.3 ((Maximal/Maximum/Perfect) Matching)

Given graph $G = (V, E)$, a set $M \subseteq E$ is a *matching* (in G) if (V, M) has max-degree 1.
disjoint pairs of vertices

M is *($\subseteq$-) maxima**l*** (a.k.a. *saturated*) if no superset of M is a matching.
M is a *maxim**um** matching* is there is no matching of strictly larger cardinality in G.
M is a *perfect matching* if $|M| = |V|/2$.

Note:

- $\subseteq$-maximal matchings easy to find via greedy algorithm.
- Maxim**um** matchings are much more complicated, but also computable in polytime (Edmonds's "Blossom algorithm")

Matching → Vertex Cover

Lemma 10.4 (VC ≥ M)

If M is a matching and C is a vertex cover in G, then $|C| \ge |M|$. ◀

Proof:

Let $\{v, w\} \in M \subseteq E$. $\rightsquigarrow$ C has to contain v or w (or both).

Since all $|M|$ matching edges are disjoint, C must cover them by $\ge |M|$ distinct endpoint. ■

```
procedure matchingVertexCoverApprox(G = (V, E))
    // greedy maximal matching
    M := ∅
    for e ∈ E // arbitrary order
        if M ∪ {e} is a matching
            M := M ∪ {e}
    return ⋃_{ {u,v}∈M } {u, v}
```

Theorem 10.5 (Matching is 2-approx for Vertex Cover)

matchingVertexCoverApprox is a *2-approximation* for VertexCover. ◀

Can we do better?

Maybe do smarter analysis?

A tight example for "VC $\geq$ M": $K_{n,n}$

Assuming the *unique games conjecture*, no polytime $(2 - \varepsilon)$ approx for VC.

Simple matching-based approximation worst-case optimal . . .

10.3 The Drosophila of Approximation: Set Cover

(Weighted) Set Cover

Definition 10.6 (SetCover)

Given: a number n, $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k subsets of $U = [n]$,
and a cost function $c : S \to \mathbb{N}$.

Solutions: $\mathcal{C} \subseteq [k]$ with $\bigcup_{i \in \mathcal{C}} S_i = U$

Cost: $\sum_{i \in \mathcal{C}} c(S_i)$

Goal: min ◀

- *cardinality version* a.k.a. UnweightedSetCover has cost $c(S) = |S|$
- UnweightedSetCover generalizes VertexCover:
  For VertexCover instances, the sets S_i are the sets of edges incident at a vertex v
  $\rightsquigarrow$ additional property that each $e \in U$ occurs in **exactly** 2 sets S_i
- general UnweightedSetCover = Vertex Cover on hypergraphs

We will use SetCover to illustrate various techniques for approximation algorithms.

Greedy Algorithm

Arguably simplest approach: **Greedily** pick set with current best *cost-per-new-item* ratio.

```
procedure greedySetCover(n, 𝒮, c)
    𝒞 := ∅;  C := ∅
    // For analysis:  i := 1
    while C ≠ [n]
        i* := arg min_{i∈[n]} c(S_i) / |S_i \ C|
        𝒞 := 𝒞 ∪ {i*}
        C := C ∪ S_{i*}
        // For analysis only:
        // α_i := c(S_{i*}) / |S_{i*} \ C|
        // for e ∈ S_{i*} \ C set price(e) := α_i
        // i := i + 1
    return 𝒞
```

Lemma 10.7 (Price Lemma)

Let $e_1, e_2, \ldots, e_n$ the order, in which greedySetCover covers the elements of U.
Then for all $j \in \{1, \ldots, n\}$ we have

$$price(e_j) \;\le\; \frac{OPT}{n-j+1}.$$

Proof:

Consider time when the jth element e_j is covered.

$|\overline{C}| = n - (j-1)$ elements uncovered (for $\overline{C} = U \setminus C$).

Optimal SC $\mathcal{C}^*$ covers $\overline{C}$ with cost $\le OPT$

$$\rightsquigarrow \quad \exists S_{i^*} \;:\; \underbrace{\frac{c(S_{i^*})}{|S_{i^*} \setminus C|}}_{\ge\, price(e_j)} \;\le\; \frac{OPT}{|\overline{C}|} \;\le\; \frac{OPT}{n-j+1}.$$

in $\mathcal{C}^*$, but not (yet) in $\mathcal{C}$

Arbitrarily order sets in $\mathcal{C}^*$, assign prices to uncovered elements.
If all prices were $> OPT/|\overline{C}|$, covering $\overline{C}$ would cost $> OPT$. ϟ

■

Greedy Set Cover Analysis

Theorem 10.8 (greedySetCover approx)

greedySetCover is an H_n-approximation for WeightedSetCover.

Proof:

$$
\begin{aligned}
c(\mathcal{C}) &= \sum_{i \in \mathcal{C}} c(S_i) = \sum_{j=1}^{n} price(e_j) \\
&\underset{[\text{Lemma } 10.7]}{\le} \sum_{j=1}^{n} \frac{OPT}{n-j+1} = OPT \sum_{i=1}^{n} \frac{1}{n} = H_n \cdot OPT \qquad \blacksquare
\end{aligned}
$$

Greedy Worst Case

$H_n \sim \ln n$ is ... not amazing. (Guarantee becomes worse with growing input size)

Unfortunately, Bound is **tight** for greedySetCover in the worst case even on WEIGHTED**VERTEXCOVER** instances:

- Consider star graph where leaves cost $\frac{1}{n}, \frac{1}{n-1}, \ldots, 1$, and middle vertex costs $1 + \varepsilon$.
- greedySetCover picks all leaves $\rightsquigarrow$ H_n
- $OPT = 1 + \varepsilon$

More complicated constructions: $\Omega(\log n)$-approx even for (UNWEIGHTED)VERTEXCOVER.

10.4 The Layering Technique for Set Cover

Size-proportional cost functions

Greedy failed on "unfair" costs for sets . . . what if costs are "nicer"?
Larger sets "should" be more costly.

Definition 10.9 (Size-proportional cost function)

A cost function c is called *size proportional* if there is a constant p so that $c(S_i) = p|S_i|$. ◀

Definition 10.10 (Frequency)

The *frequency* f_e of an element $e \in [n]$ is the number of sets in which it occurs:
$f_e = |\{j : e \in S_j\}|$.
The (maximal) *frequency* of a SetCover instance is $f = \max_e f_e$. ◀

Note: (Weighted)VertexCover instance $\rightsquigarrow$ $f = 2$

Size-proportional indeed easier

Lemma 10.11 (size-proportionality → trivial f-approx)

For a size proportional weight function c we have $c(\mathcal{S}) \le f \cdot OPT$.

Proof:

$$c(\mathcal{S}) = \sum_{i=1}^{k} c(S_i) = p \sum_{i=1}^{k} |S_i| = p \sum_{e \in U} f_e \le p \sum_{e \in U} f \overset{\text{size-prop.} \rightsquigarrow OPT \ge p \cdot n}{\le} f \cdot OPT$$

■

Taking *all* sets gives f-approx, so certainly true for greedySetCover.

But probably not too many problem instances are that simple . . .

Layering Algorithm

Idea: Split cost function into sum of

- size-proportional part c_0 and
- a some residue c_1

---

procedure layeringSetCover(U, $\mathcal{S}$, c)
    $p := \min\left\{\dfrac{c(S_j)}{|S_j|} : j \in [k]\right\}$
    $c_0(S_i) := p \cdot |S_i|$ // *size-prop. part*
    $c_1(S_i) := c(S_i) - c_0(S_i)$ // ≥ 0
    $\mathcal{C}_0 := \{j \in [k] : c_1(S_j) = 0\}$
    $U_0 := \bigcup_{j \in \mathcal{C}_0} S_j$ // *covered by size-prop.*
    if $U_0 == U$
        return $\mathcal{C}_0$
    else
        $U_1 := U \setminus U_0$ // *rest of universe*
        $\mathcal{S}_1 := \{S \in \{S_1, \ldots, S_k\} \mid S \cap U_1 \neq \emptyset\}$
        $\mathcal{C}_1 :=$ layeringSetCover(U_1, $\mathcal{S}_1$, c_1)
        return $\mathcal{C}_0 \cup \mathcal{C}_1$

Theorem 10.12 (layering f-approx)

layeringSetCover is f-approx. for SetCover. ◀

Proof:
Show by induction over recursive calls that (a) compute cover (b) cost $\leq f \cdot OPT$.

Basis: $U_1 = \emptyset$
All of U covered by size-prop. part/
$\rightsquigarrow$ f-approx by Lemma 10.11

Inductive step:
IH: $\mathcal{C}_1$ covers U_1 at cost $c_1(\mathcal{C}_1) \leq f \cdot OPT(U_1, \mathcal{S}_1, c_1)$.
Let $\mathcal{C}^*$ be **optimal** set cover w.r.t. c

Lemma 10.11: $\mathcal{C} = \mathcal{C}_0 \cup \mathcal{C}_1$ is f-approx w.r.t. $c_\mathbf{0}$.
$\rightsquigarrow$ $c_0(\mathcal{C}) \leq f \cdot c_0(\mathcal{C}^*)$ (0)

Layering Algorithm [2]

Proof (cont.):

Define $\mathcal{C}_1^* = \{i \in \mathcal{C}^* : S_i \in \mathcal{S}_1\}$

$\mathcal{C}_1^*$ is a set cover for U_1

$$\rightsquigarrow \quad c_1(\mathcal{C}_1) \underset{IH}{\leq} OPT(U_1, \mathcal{S}_1, c_1) \leq f \cdot c_1(\mathcal{C}_1^*) \qquad (1)$$

$$\begin{aligned} c(\mathcal{C}) &= c_0(\mathcal{C}) + c_1(\mathcal{C}) \\ &\underset{\substack{\uparrow \\ i \in \mathcal{C}_0 \rightsquigarrow c_1 = 0}}{=} c_0(\mathcal{C}) + c_1(\mathcal{C}_{\mathbf{1}}) \\ &\underset{(0),(1)}{\leq} f \cdot \big(c_0(\mathcal{C}^*) + c_1(\mathcal{C}_1^*)\big) \\ &\leq f \cdot \big(c_0(\mathcal{C}^*) + c_1(\mathcal{C}^*)\big) \\ &= f \cdot c(\mathcal{C}^*) \end{aligned}$$

$\blacksquare$

Note: For VertexCover, this yields again a 2-approximation.

$\rightsquigarrow$ Same as using maximal matching

But the layering algorithm can handle arbitrary vertex costs (WeightedVertexCover)!

10.5 Applications of Set Cover

Shortest Superstrings

Definition 10.13 (ShortestSuperstring)

Given: alphabet Σ, set of strings $W = \{w_1, \ldots, w_n\} \subseteq \Sigma^+$

Feasible Instances: *superstrings* s of S, i. e., s contains w_i as substring for $1 \le i \le n$.

Cost: $|s|$

Goal: min ◀

Remark 10.14

Without-loss-of-generality assumption: no string is a substring of another. ◀

- Motivation: DNA assembly (sequencing from many shorter "reads")
- General problem is NP-complete

Here: Reduce this problem to SetCover!

Shortest Superstring by Set Cover

Construct *all pairwise* superstrings: overlap w_i and w_j by exactly ℓ characters (if possible)

$\sigma_{i,j,\ell} = w_i[0..|w_i|-\ell) \cdot w_j$ valid iff $w_j[0..\ell) = w_i[|w_i|-\ell..|w_i|)$

$M = \left\{\sigma_{i,j,\ell} : i,j \in [u], \ell \in \left[0..\min\{|w_i|,|w_j|\}\right]\right\}$

$\rightsquigarrow$ **Set Cover instance:**

- **Universe:** $[n]$ $\rightsquigarrow$ try to *cover* all words in W with superstring ...
- **Subsets:** $S = \{S_\pi : \pi \in W \cup M\}$ ... by combining pairwise superstrings.
  where $S_\pi = \{k \in [n] \ : \ \exists i,j : w_k = \pi[i..j)\}$
- **Cost function:** $c(S_\pi) = |\pi|$

Given set-cover solution $\{S_{\pi_1}, \ldots, S_{\pi_k}\}$
$\rightsquigarrow$ superstring $s = \pi_1 \ldots \pi_k$ (in any order)

Shortest Superstring by Set Cover – Analysis

Lemma 10.15 (Pairwise superstrings yield 2-SC-approx)

Let W be an instance for ShortestSuperstring and (n, S, c) the corresponding SetCover instance. Let further OPT resp. OPT_{SC} be the optimal objective value of W resp. (n, S, c). Then $OPT \leq OPT_{SC} \leq 2 \cdot OPT$. ◀

Corollary 10.16 ($2H_n$ approximation for superstring)

By solving the transformed set cover instance with greedySetCover, we obtain a $2H_n$-approximation for the shortest superstring problem. ◀

Proof (Lemma 10.15):

- "$OPT \leq OPT_{SC}$"
  It suffices to show that $s = \pi_1 \ldots \pi_k$ is a valid superstring.
  By definition, every w_i must be contained in some π_k as a substring
- "$OPT_{SC} \leq 2 \cdot OPT$"
  $OPT = |s^*|$ for a *shortest* superstring s^* for W.
  Without loss of generality, suppose s^* contains $w_1, \ldots, w_n$ *in this order*.

Shortest Superstring by Set Cover – Analysis [2]

Proof:

Define groups: $i_1 = 1$; $i_j = \min\{i > i_{j-1}$: first occurrence of w_i does not overlap $w_{i_{j-1}}\}$.

10.6 (F)PTAS: Arbitrarily Good Approximations

Approximation Schemes

Definition 10.17

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ an optimization problem.

An algorithm A is called *polynomial time approximation scheme (PTAS)* for U, if A computes for each pair $(x, \varepsilon) \in L_I \times \mathbb{R}^+$ a feasible solution which is at most a factor $(1 + \varepsilon)$ worse than the optimum (i. e., ε is the relative error) and needs a polynomial time in $|x|$ (i. e., $O(|x|^{\exp(1/\varepsilon)})$ is possible).

If the running time of A is polynomially bounded in $|x|$ and ε^{-1}, A is called a *fully polynomial time approximation scheme (FPTAS)* for U. ◀

FPTAS for Knapsack

Assumption: any item fits in the knapsack alone, i. e., $w_i \leq b$

---

procedure approxKnapsack(w,v,b,ε)
    $\hat{V} = \max_{i=1,\ldots,n} v_i$
    $K = \varepsilon \hat{V}/n$
    $\tilde{v} = \lfloor \frac{v}{K} \rfloor$
    return DPKnapsack(w,$\tilde{v}$,b)

Theorem 10.18

`approxKnapsack` is an FPTAS for 0/1-Knapsack ◀

FPTAS asks for much

Theorem 10.19 (FPTAS → FPT and pseudopolynomial)

1. $U \in \mathsf{FPTAS} \implies p\text{-}U \in \mathsf{FPT}$
2. $U \in \mathsf{FPTAS}$ and $cost(u, x) < p\big(MaxInt(x)\big)$ for some polynomial p
   $\implies \exists$ pseudopolynomial algorithm for U.

10.7 Christofides's Algorithm

Metric TSP

MST Approx

Matching and Triangle Inequality

10.8 Randomized Approximations

Randomized Approximation Guarantees

Definition 10.20 (Randomized δ-approx.)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, \mathbf{max})$ an optimization problem. For $\delta > 1$ a randomized algorithm A is called *randomized δ-approximation algorithm for U*, if

- $\mathbb{P}[A(x) \in M(x)] \;=\; 1,$ (always feasible) and
- $\mathbb{P}[R_A(x) \leq \delta] \;\geq\; \frac{1}{2}$ (typically within δ)

for all $x \in L_I$. ◀

Definition 10.21 (δ-expected approx.)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, \max)$ an optimization problem. For $\delta > 1$ a randomized algorithm A is called *(randomized) δ-expected approximation algorithm for U*, if

- $\mathbb{P}[A(x) \in M(x)] \;=\; 1$ (always feasible) and
- $\dfrac{\mathbb{E}[cost(A(x))]}{OPT_U(x)} \;\leq\; \delta$ (expected within δ)

for all $x \in L_I$. ◀

(Minimization problems similar.)

Randomized Max-Sat Approximation

Recall: k-Max-Sat asks for an assignment satisfying a maximal number of clauses.
Assumption: Each clause contains *exactly* k literals over k different variables.

---

procedure randomAssignment(φ)
    Let φ have variables $x_1, \ldots, x_n$
    Choose assignment $\alpha \in \{0,1\}^n$ uniformly at random
    s = number of clauses in φ satisfied by α
    return (s, α)

Theorem 10.22 (randomAssignment is approx)

randomAssignment is

1. a $\frac{2^k}{2^k-1}$-expected approximation and
2. a randomized $\frac{2^{k-1}}{2^{k-1}-1}$-approximation

for k-Max-Sat.