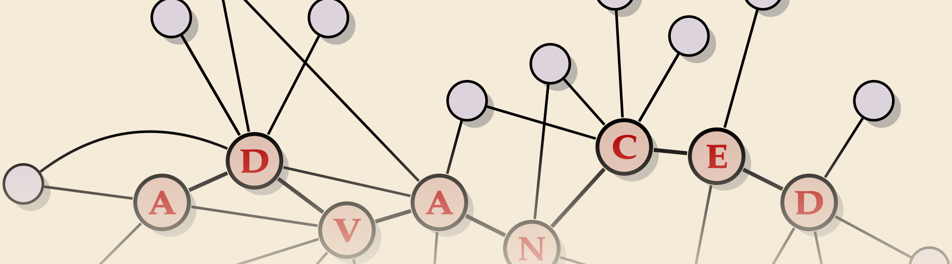


10 Approximation Algorithms

8 July 2025

Prof. Dr. Sebastian Wild

CS627 (Summer 2025)
Philipps-Universität Marburg
version `2025-07-14 21:09`

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 $cost(y) \in \mathbb{Q}$

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 . . .)

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"*?

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. $A(x) \in M(x)$

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

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

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*** $\boldsymbol{R_A(x)}$ ***of*** $\boldsymbol{A}$ ***on*** $\boldsymbol{x}$ 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 $\forall x \in L_I : R_A(x) \geq c$.

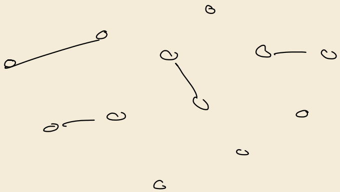
10.2 Vertex Cover and Matchings

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?

Need a way to lower bound OPT!

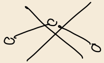
Example: Vertex Cover



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 *(⊆-) maxim**al*** (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 $|M| = |V|/2$.

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 *(⊆-) maxim**al*** (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:

- ⊆-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 $\geq$ M)

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

Matching → Vertex Cover

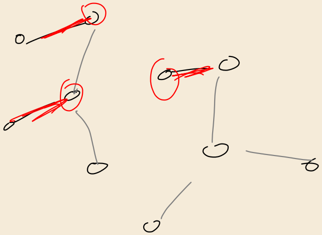
Lemma 10.4 (VC ≥ M)
If M is a matching and C is a vertex cover in G, then $|C| \geq |M|$.

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

Matching → Vertex Cover

Lemma 10.4 (VC ≥ M)
If M is a matching and C is a vertex cover in G, then $|C| \geq |M|$.

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 $\geq |M|$ distinct endpoint. ∎



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}
```

Matching → Vertex Cover

Lemma 10.4 (VC ≥ M)

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

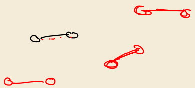
Proof:

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 $\geq |M|$ 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}   = C
```

(1) C is VC

otherwise some edge uncovered



(2) 2-approx

$|C| = 2|M|$

$\frac{|C|}{OPT} \leq \frac{2|M|}{|M|} = 2$

$OPT \geq |M|$

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

matchingVertexCoverApprox is a *2-approximation* for VertexCover.

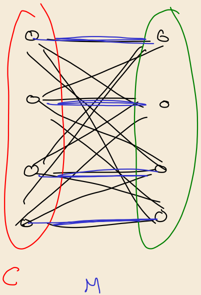
Can we do better?

Maybe do smarter analysis?

Can we do better?

Maybe do smarter analysis?

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



Can we do better?

Maybe do smarter analysis?

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

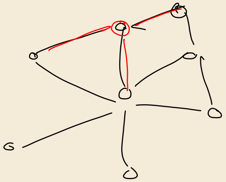
$\varepsilon > 0$ constant

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 $c(S) = \cancel{|S|}$ 1
- 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

(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, $\mathcal{S}$, c)
  $\mathcal{C} := \emptyset$; $C := \emptyset$
  // *For analysis:* $i := 1$
  while $C \neq [n]$ //assume $\bigcup S_i = [n]$
    $i^* := \arg\min_{i \in [n]} \dfrac{c(S_i)}{|S_i \setminus C|}$
    $\mathcal{C} := \mathcal{C} \cup \{i^*\}$
    $C := C \cup S_{i^*}$
    // *For analysis only:*
    // $\alpha_i := \dfrac{c(S_{i^*})}{|S_{i^*} \setminus C|}$
    // *for* $e \in S_{i^*} \setminus C$ *set* $price(e) := \alpha_i$
    // $i := i + 1$
  return $\mathcal{C}$

Greedy Algorithm

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

procedure greedySetCover$(n, \mathcal{S}, c)$
  $\mathcal{C} := \emptyset;\quad C := \emptyset$
  // *For analysis:* $i := 1$
  while $C \neq [n]$
    $i^* := \arg\min_{i \in [n]} \dfrac{c(S_i)}{|S_i \setminus C|}$
    $\mathcal{C} := \mathcal{C} \cup \{i^*\}$
    $C := C \cup S_{i^*}$
    // *For analysis only:*
    // $\alpha_i := \dfrac{c(S_{i^*})}{|S_{i^*} \setminus C|}$
    // *for* $e \in S_{i^*} \setminus C$ *set* $price(e) := \alpha_i$
    // $i := i + 1$
  return $\mathcal{C}$

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

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

Proof:
Consider time when the jth element e_j is covered.

Greedy Algorithm

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

procedure greedySetCover$(n, \mathcal{S}, c)$
  $\mathcal{C} := \emptyset$; $C := \emptyset$
  // *For analysis:* $i := 1$
  while $C \neq [n]$
    $i^* := \arg\min_{i \in [n]} \dfrac{c(S_i)}{|S_i \setminus C|}$
    $\mathcal{C} := \mathcal{C} \cup \{i^*\}$
    $C := C \cup S_{i^*}$
    // *For analysis only:*
    // $\alpha_i := \dfrac{c(S_{i^*})}{|S_{i^*} \setminus C|}$
    // *for* $e \in S_{i^*} \setminus C$ *set* $price(e) := \alpha_i$
    // $i := i + 1$
  return $\mathcal{C}$

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

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

Proof:
Consider time when the jth element e_j is covered.
$|\overline{C}| = n - (j - 1)$ elements uncovered (for $\overline{C} = U \setminus C$).

Greedy Algorithm

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

procedure greedySetCover($n, \mathcal{S}, c$)
  $\mathcal{C} := \emptyset$; $C := \emptyset$
  // For analysis: $i := 1$
  while $C \neq [n]$
    $i^* := \arg\min_{i \in [n]} \dfrac{c(S_i)}{|S_i \setminus C|}$
    $\mathcal{C} := \mathcal{C} \cup \{i^*\}$
    $C := C \cup S_{i^*}$
    // For analysis only:
    // $\alpha_i := \dfrac{c(S_{i^*})}{|S_{i^*} \setminus C|}$
    // *for* $e \in S_{i^*} \setminus C$ *set* $price(e) := \alpha_i$
    // $i := i + 1$
  return $\mathcal{C}$

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

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

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 $\leq OPT$

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

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

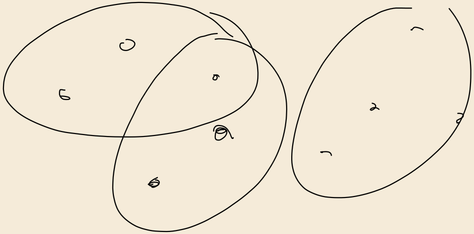
$H_n = \ln n + \overset{0.57..}{\gamma} + o(1)$

Theorem 10.8 (greedySetCover approx)

greedySetCover is an H_n-approximation for WEIGHTEDSETCOVER.

Proof:

$$c(\mathscr{C}) \;=\; \sum_{i \in \mathscr{C}} c(S_i) \;=\; \sum_{j=1}^{n} price(e_j)$$



Greedy Set Cover Analysis

Theorem 10.8 (greedySetCover approx)

greedySetCover is an H_n-approximation for WEIGHTEDSETCOVER.

Proof:

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

$\blacksquare$

Greedy Worst Case

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

Greedy Worst Case

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

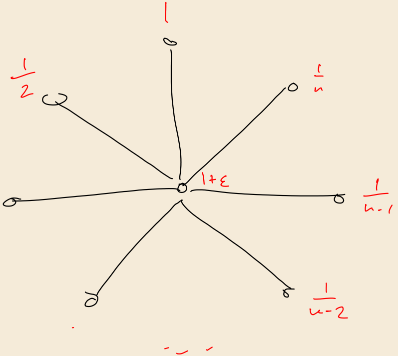
Unfortunately, bound is **tight** for greedySetCover in the worst case even on WEIGHTEDVERTEXCOVER instances:

- 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 $\rightsquigarrow H_n$
- $OPT = 1 + \varepsilon$

greedy step 1

center $= \frac{1+\varepsilon}{n}$

leaf $= \frac{1}{n}$



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 WEIGHTEDVERTEXCOVER 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
- $OPT = 1 + \varepsilon$

for greedySetCover

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.

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 $c(S_i) = p|S_i|$. ◀

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:

$$c(\mathcal{S}) = \sum_{i=1}^{k} c(S_i) = p \sum_{i=1}^{k} |S_i|$$

Size-proportional indeed easier

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

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

Proof:

$$c(\mathcal{S}) = \sum_{i=1}^{k} c(S_i) = p \sum_{i=1}^{k} |S_i| = p \sum_{e \in U} f_e \leq p \sum_{e \in U} f$$

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:

size-prop. $\rightsquigarrow$ $OPT \ge p \cdot n$

$$c(\mathcal{S}) = \sum_{i=1}^{k} c(S_i) = p \sum_{i=1}^{k} |S_i| = p \sum_{e \in U} f_e \le p \sum_{e \in U} f \le f \cdot OPT \qquad \blacksquare$$

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:

size-prop. $\rightsquigarrow$ $OPT \ge p \cdot n$

$$c(\mathcal{S}) = \sum_{i=1}^{k} c(S_i) = p \sum_{i=1}^{k} |S_i| = p \sum_{e \in U} f_e \le p \sum_{e \in U} f \le f \cdot OPT \qquad \blacksquare$$

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

$$c(S_i) = \underset{p \cdot |S_i|}{\underset{\parallel}{c_0(S_i)}} + \underbrace{c_1(S_i)}_{\geq 0}$$

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
    $\mathcal{C}_0 := \{j \in [k] : \underline{c_1(S_j) = 0}\}$
    $U_0 := \bigcup_{j \in \mathcal{C}_0} S_j$ *// 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 $\mathcal{C}_0 \cup \mathcal{C}_1$

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. ◀

Layering Algorithm

Idea: Split cost function into sum of

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

---

procedure layeringSetCover($U, \mathcal{S}, c$)
    $p := \min\left\{ \frac{c(S_j)}{|S_j|} : j \in [k] \right\}$
    $c_0(S_i) := p \cdot |S_i|$ *// 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) computes cover (b) of cost $\leq f \cdot OPT$.

Basis: $U_0 = U$
All of U covered by size-prop. part ✓ (a) ✓
$\rightsquigarrow$ f-approx by Lemma 10.11 (b)

/

on restricted instance

$S' = C_0$

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)
    $p := \min\left\{\frac{c(S_j)}{|S_j|} : j \in [k]\right\}$
    $c_0(S_i) := p \cdot |S_i|$ *// 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) computes cover (b) of cost $\leq f \cdot OPT$.

Basis: $U_0 = U$
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 $c_1(\mathcal{C}_1) \leq f \cdot OPT(U_1, \mathcal{S}_1, c_1)$.

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)
 $p := \min\left\{ \frac{c(S_j)}{|S_j|} : j \in [k] \right\}$
 $c_0(S_i) := p \cdot |S_i|$ *// 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) computes cover (b) of cost $\leq f \cdot OPT$.

Basis: $U_0 = U$
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

$c(C^*) = OPT$

Layering Algorithm

Idea: Split cost function into sum of

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

---

procedure layeringSetCover($U, \mathcal{S}, c$)
  $p := \min\left\{\frac{c(S_j)}{|S_j|} : j \in [k]\right\}$
  $c_0(S_i) := p \cdot |S_i|$ *// 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) computes cover (b) of cost $\leq f \cdot OPT$.

Basis: $U_0 = U$
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

size-prop.

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

Layering Algorithm [2]

Proof (cont.):

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

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 — all S_i with $i \in \mathcal{C}^* \setminus \mathcal{C}_1^*$ only U_0

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

$$OPT(U_1, S_1, c_1) \leq c_1(\mathcal{C}_1^*)$$

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

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

$$\rightsquigarrow \quad c_0(\mathcal{C}) \leq f \cdot c_0(\mathcal{C}^*) \qquad (0)$$

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

∎

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

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

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

∎

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: $|s|$

$s[j_i..j_i+|w_i|) = w_i$

Goal: min

Remark 10.14

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

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

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

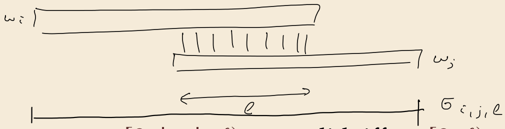
Without-loss-of-generality assumption: no string is a substring of another. ◀

- 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)



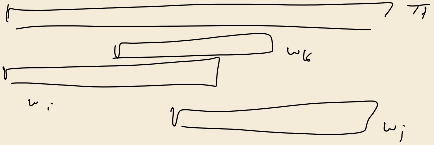
$\sigma_{i,j,\ell} = w_i[0..|w_i|-\ell) \cdot w_j$ valid iff $w_j[0..\ell) = w_i[|w_i|-\ell..|w_i|)$

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

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

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

w_k occurs as substring in π



$i, j, k \in S_\pi$

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 $OPT \le OPT_{SC} \le 2 \cdot OPT$. ◀

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 $OPT \le OPT_{SC} \le 2 \cdot OPT$. ◀

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. ◀

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.

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 $OPT \le OPT_{SC} \le 2 \cdot OPT$.

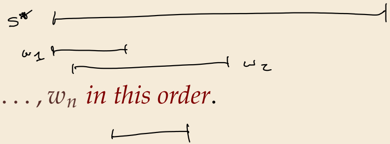
Corollary 10.16 ($2H_n$ approximation for superstring)

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

Proof (Lemma 10.15):

- "$OPT \le OPT_{SC}$"
  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.

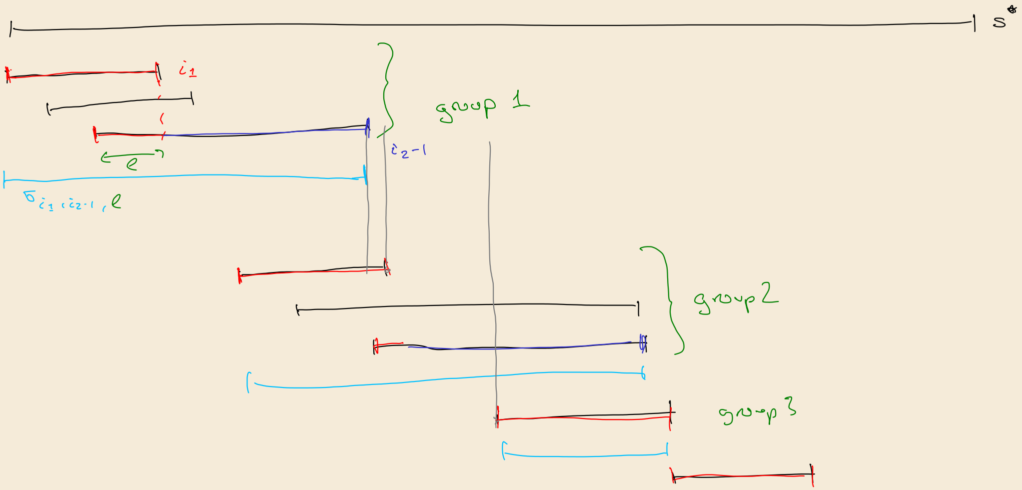
- "$OPT_{SC} \le 2 \cdot OPT$"
  $OPT = |s^*|$ for a *shortest* superstring s^* for W.
  Without loss of generality, suppose s^* contains $w_1, \ldots, w_n$ *in this order*.



Shortest Superstring by Set Cover – Analysis [2]

Proof:

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



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 $w_{i_{j-1}}\}$.

Group j starts with w_{i_j} and ends with $w_{i_{j+1}-1}$

$\rightsquigarrow$ overlap of two strings $\rightsquigarrow$ $\pi_j = \sigma_{i_j, i_{j+1}-1, \ell_j}$

Shortest Superstring by Set Cover – Analysis [2]

Proof:

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

Group j starts with w_{i_j} and ends with $w_{i_{j+1}-1}$

$\rightsquigarrow$ overlap of two strings $\rightsquigarrow$ $\pi_j = \sigma_{i_j, i_{j+1}-1, \ell_j}$

Groups can overlap (so concatenation of σs longer than s^*).

Shortest Superstring by Set Cover – Analysis [2]

Proof:

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

Group j starts with w_{i_j} and ends with $w_{i_{j+1}-1}$

$\rightsquigarrow$ overlap of two strings $\rightsquigarrow$ $\pi_j = \sigma_{i_j, i_{j+1}-1, \ell_j}$

Groups can overlap (so concatenation of σs longer than s^*).

But group j and $j + \mathbf{2}$ cannot overlap!

$\rightsquigarrow$ $|\pi_1 \dots \pi_k| \le 2|s^*| = 2 \cdot OPT$. ∎

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 $w_{i_{j-1}}\}$.

Group j starts with w_{i_j} and ends with $w_{i_{j+1}-1}$

$\rightsquigarrow$ overlap of two strings $\rightsquigarrow$ $\pi_j = \sigma_{i_j, i_{j+1}-1, \ell_j}$

Groups can overlap (so concatenation of σs longer than s^*).

But group j and $j + \mathbf{2}$ cannot overlap!

$\rightsquigarrow$ $|\pi_1 \ldots \pi_k| \le 2|s^*| = 2 \cdot OPT.$ ∎

(Note: Better approximation algorithms for ShortestSuperstring possible via different techniques.)

10.6 (F)PTAS: Arbitrarily Good Approximations

Approximation Schemes

The problems so far had a barrier to arbitrarily good approximations;
but sometimes we can achieve the latter!

Approximation Schemes

The problems so far had a barrier to arbitrarily good approximations;
but sometimes we can achieve the latter!

$\varepsilon \mapsto \mathrm{Time}_{A_\varepsilon}(c)$

FPTAS polynomial

Definition 10.17 ((F)PTAS)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, \mathbf{min})$ an optimization problem.

An algorithm $A = A_\varepsilon(x)$ with input (ε, x) is called ***polynomial-time approximation scheme (PTAS)*** for U, if for every *constant* $\varepsilon \in \mathbb{Q}_{>0}$, the algorithm A_ε is a $(1+\varepsilon)$-approximation for U with running time polynomial in $|x|$.

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

Note: PTAS could have running time $O(n^{2^{1/\varepsilon}})$ or so (akin to XP ~~running~~ XP time)

$O(n^{2^{1/\varepsilon}})$

Approximation Schemes

The problems so far had a barrier to arbitrarily good approximations;
but sometimes we can achieve the latter!

Definition 10.17 ((F)PTAS)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, \textbf{min})$ an optimization problem.

An algorithm $A = A_\varepsilon(x)$ with input (ε, x) is called
polynomial-time approximation scheme (PTAS) for U,
if for every *constant* $\varepsilon \in \mathbb{Q}_{>0}$, the algorithm A_ε is a $(1+\varepsilon)$-approximation for U with running time polynomial in $|x|$.

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

Note: PTAS could have running time $O(n^{2^{1/\varepsilon}})$ or so (akin to XP running time)

FPTAS much stronger

Approximation Schemes

The problems so far had a barrier to arbitrarily good approximations;
but sometimes we can achieve the latter!

Definition 10.17 ((F)PTAS)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, \textbf{min})$ an optimization problem.

An algorithm $A = A_\varepsilon(x)$ with input (ε, x) is called ***polynomial-time approximation scheme (PTAS)*** for U, if for every *constant* $\varepsilon \in \mathbb{Q}_{>0}$, the algorithm A_ε is a $(1 + \varepsilon)$-approximation for U with running time polynomial in $|x|$.

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

Note: PTAS could have running time $O(n^{2^{1/\varepsilon}})$ or so (akin to XP running time)

FPTAS much stronger … *but do they even exist for any* NP*-hard problems?*

Approximation Schemes

The problems so far had a barrier to arbitrarily good approximations;
but sometimes we can achieve the latter!

Definition 10.17 ((F)PTAS)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, \mathbf{min})$ an optimization problem.

An algorithm $A = A_\varepsilon(x)$ with input (ε, x) is called
polynomial-time approximation scheme (PTAS) for U,
if for every *constant* $\varepsilon \in \mathbb{Q}_{>0}$, the algorithm A_ε is a $(1+\varepsilon)$-approximation for U with running time polynomial in $|x|$.

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

Note: PTAS could have running time $O(n^{2^{1/\varepsilon}})$ or so (akin to XP running time)

FPTAS much stronger ... *but do they even exist for any* NP*-hard problems?* Yes!

Pseudopolynomial DP Reprise

Recall **0/1-Knapsack**: **Given:** items $1, \ldots, n$ with *weights* $w_1, \ldots, w_n$ and *values* $v_1, \ldots, v_n$;
Feasible solutions: subset of items with total weight $\leq b$
Goal: maximize total value

Approximation Idea: Work with *rounded* values (depending on ε)

Pseudopolynomial DP Reprise

Recall **0/1-Knapsack**: **Given:** items $1, \ldots, n$ with *weights* $w_1, \ldots, w_n$ and *values* $v_1, \ldots, v_n$;
Feasible solutions: subset of items with total weight $\le b$
Goal: maximize total value

Approximation Idea: Work with *rounded* values (depending on ε)

In Unit 3, we solved Knapsack

- using a DP table $V[n', b']$ = max value from items $1..n'$ and total weight $b' \le b$

$\rightsquigarrow$ $n \cdot b$ entries $\rightsquigarrow$ total time $O(n \cdot b \cdot \underline{\log(MaxInt(v))})$ $\le |x|$

$\rightsquigarrow$ good if *weights* are small, but we want to round *values*

Pseudopolynomial DP Reprise

Recall **0/1-Knapsack**: **Given:** items $1, \ldots, n$ with *weights* $w_1, \ldots, w_n$ and *values* $v_1, \ldots, v_n$;
Feasible solutions: subset of items with total weight $\leq b$
Goal: maximize total value

Approximation Idea: Work with *rounded* values (depending on ε)

In Unit 3, we solved Knapsack

- using a DP table $V[n', b']$ = max value from items $1..n'$ and total weight $b' \leq b$

$\rightsquigarrow$ $n \cdot b$ entries $\rightsquigarrow$ total time $O(n \cdot b \cdot \log(MaxInt(v)))$

$\rightsquigarrow$ good if *weights* are small, but we want to round *values*

- actually, DP also works with values as index!

  $\boxed{\text{Assumption: } w_1, \ldots, w_n, v_1, \ldots, v_n \in \mathbb{N}}$!

- DP table $W[n', v]$ = min weight from items $\underline{1, \ldots, n'}$ with value $\underline{\underline{=v}}$

Pseudopolynomial DP Reprise

Recall **0/1-Knapsack**: **Given:** items $1, \ldots, n$ with *weights* $w_1, \ldots, w_n$ and *values* $v_1, \ldots, v_n$;
Feasible solutions: subset of items with total weight $\le b$
Goal: maximize total value

Approximation Idea: Work with *rounded* values (depending on ε)

In Unit 3, we solved Knapsack

- using a DP table $V[n', b']$ = max value from items $1..n'$ and total weight $b' \le b$

$\rightsquigarrow$ $n \cdot b$ entries $\rightsquigarrow$ total time $O(n \cdot b \cdot \log(MaxInt(v)))$

$\rightsquigarrow$ good if *weights* are small, but we want to round *values*

- actually, DP also works with values as index!

$\boxed{\text{Assumption: } w_1, \ldots, w_n,\ v_1, \ldots, v_n \in \mathbb{N}}$

- DP table $W[n', v]$ = min weight from items $1, \ldots, n'$ with value $= v$

$$W[n', v] = \begin{cases} \min\left\{W[n'-1, v],\ \underline{W[n'-1, v - v_{n'}] + w_{n'}}\right\} & \text{if } v_{n'} < v \\ W[n'-1, v] & \text{otherwise} \end{cases}$$

(+ initial values)

Pseudopolynomial DP Reprise

Recall **0/1-Knapsack**: **Given:** items $1, \ldots, n$ with *weights* $w_1, \ldots, w_n$ and *values* $v_1, \ldots, v_n$;
Feasible solutions: subset of items with total weight $\le b$
Goal: maximize total value

Approximation Idea: Work with *rounded* values (depending on ε)

In Unit 3, we solved Knapsack

- using a DP table $V[n', b']$ = max value from items $1..n'$ and total weight $b' \le b$

$\rightsquigarrow$ $n \cdot b$ entries $\rightsquigarrow$ total time $O(n \cdot b \cdot \log(MaxInt(v)))$

$\rightsquigarrow$ good if *weights* are small, but we want to round *values*

- actually, DP also works with values as index!

$\boxed{\text{Assumption: } w_1, \ldots, w_n,\ v_1, \ldots, v_n \in \mathbb{N}}$

- DP table $W[n', v]$ = min weight from items $1, \ldots, n'$ with value $= v$

$$W[n', v] \;=\; \begin{cases} \min\Big\{W[n'-1, v],\ W[n'-1, v - v_{n'}] + w_{n'}\Big\} & \text{if } v_{n'} < v \\ W[n'-1, v] & \text{otherwise} \end{cases}$$

(+ initial values)

$\rightsquigarrow$ $n \cdot nV$ entries for $V = \max v_i$ $\rightsquigarrow$ total time $\underline{O(n^2 \cdot V \cdot \log(MaxInt(w)))}$

FPTAS for Knapsack

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

$(1-\varepsilon)$-approx

```
procedure knapsackFPTAS(w, v, b, ε)
    V := max_{i=1,...,n} v_i
    K := εV/n
    ṽ := ⌊v/K⌋  // rounded v   w/ unit K
    return DPKnapsack(w, ṽ, b)
```

DPKnapsack is pseudopolynomial DP algorithm with running time $O(n^2 \cdot V \cdot \log(\mathit{MaxInt}(w)))$

Theorem 10.18

approxKnapsack is an FPTAS for 0/1-Knapsack.

FPTAS for Knapsack

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

```
procedure knapsackFPTAS(w, v, b, ε)
    V := max_{i=1,...,n} v_i
    K := εV/n
    ṽ := ⌊v/K⌋ // rounded v
    return DPKnapsack(w, ṽ, b)
```

DPKnapsack is pseudopolynomial DP algorithm with running time $O(n^2 \cdot V \cdot \log(MaxInt(w)))$

Theorem 10.18

approxKnapsack is an FPTAS for 0/1-Knapsack.

Proof:

First consider running time; dominated by DPKnapsack.

FPTAS for Knapsack

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

```
procedure knapsackFPTAS(w, v, b, ε)
    V := max_{i=1,...,n} v_i
    K := εV/n
    ṽ := ⌊v/K⌋ // rounded v
    return DPKnapsack(w, ṽ, b)
```

DPKnapsack is pseudopolynomial DP algorithm with running time $O(n^2 \cdot V \cdot \log(MaxInt(w)))$

Theorem 10.18

approxKnapsack is an FPTAS for 0/1-KNAPSACK.

$\tilde{V} := \max \tilde{v}_i$

Proof:

First consider running time; dominated by DPKnapsack.

$$O(n^2\tilde{V}\,\underline{\log(MaxInt(w))}) \;\leq\; O(n^2\tilde{V}|x|) \;\leq\; O\left(n^2|x|\frac{V}{K}\right) \;\leq\; O\left(n^3|x|\varepsilon^{-1}\right) \;\leq\; O\left(|x|^4\varepsilon^{-1}\right)$$

FPTAS for Knapsack

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

```
procedure knapsackFPTAS(w, v, b, ε)
    V := max_{i=1,...,n} v_i
    K := εV/n
    ṽ := ⌊v/K⌋ // rounded v
    return DPKnapsack(w, ṽ, b)
```

DPKnapsack is pseudopolynomial DP algorithm with running time $O(n^2 \cdot V \cdot \log(MaxInt(w)))$

Theorem 10.18

approxKnapsack is an FPTAS for 0/1-Knapsack.

Proof:

First consider running time; dominated by DPKnapsack.

$$O(n^2 \tilde{V} \log(MaxInt(w))) \;\le\; O(n^2 \tilde{V} |x|) \;\le\; O\left(n^2 |x| \frac{V}{K}\right) \;\le\; O\left(n^3 |x| \varepsilon^{-1}\right) \;\le\; O\left(|x|^4 \varepsilon^{-1}\right)$$

It remains to show that total value of $I = \mathrm{DPKnapsack}(w, \tilde{v}, b)$ is $v(I) \ge (1 - \varepsilon) \cdot OPT$

FPTAS for Knapsack [2]

Proof (cont.):

Let I^* be an optimal solution, $v(I^*) = \sum_{i \in I^*} v_i = OPT$

For each $i \in [n]$, we have by definition $\boxed{v_i - K \;<\; K \cdot \tilde{v}_i \;\le\; v_i \quad (*)}$.

$x - 1 \le \lfloor x \rfloor \le x$

```
procedure knapsackFPTAS(w, v, b, ε)
    V := max_{i=1,...,n} v_i
    K := εV/n
    ṽ := ⌊v/K⌋ // rounded v
    return DPKnapsack(w, ṽ, b)
```

FPTAS for Knapsack [2]

Proof (cont.):

Let I^* be an optimal solution, $v(I^*) = \sum_{i \in I^*} v_i = OPT$

For each $i \in [n]$, we have by definition $\boxed{v_i - K \ < \ K \cdot \tilde{v}_i \ \leq \ v_i \quad (*)}$.

D
~~F~~PKnapsack returns *optimal* solution for rounded values $\rightsquigarrow$ $\tilde{v}(I) \ \geq \ \tilde{v}(I^*) \quad (o)$

Moreover, $OPT \ \geq \ V$ by our assumption that each item fits into knapsack. $\quad (V)$

FPTAS for Knapsack [2]

Proof (cont.):

Let I^* be an optimal solution, $v(I^*) = \sum_{i \in I^*} v_i = OPT$

For each $i \in [n]$, we have by definition $\boxed{v_i - K \;<\; K \cdot \tilde{v}_i \;\le\; v_i \quad (*)}$.

FPKnapsack returns *optimal* solution for rounded values $\rightsquigarrow$ $\tilde{v}(I) \;\ge\; \tilde{v}(I^*) \quad (o)$

Moreover, $OPT \;\ge\; V$ by our assumption that each item fits into knapsack. $\quad (V)$

We now have

$$v(I) \underset{(*)}{\ge} K \cdot \tilde{v}(I) \underset{(o)}{\ge} K \cdot \tilde{v}(I^*) \underset{(*)}{\ge} v(I^*) - nK = OPT - \varepsilon V \underset{(V)}{\ge} (1-\varepsilon) \cdot OPT$$

---

procedure knapsackFPTAS(w, v, b, ε)
  $V := \max_{i=1,\ldots,n} v_i$
  $K := \varepsilon V / n$
  $\tilde{v} := \left\lfloor \frac{v}{K} \right\rfloor$ // *rounded* v
  return DPKnapsack($w, \tilde{v}, b$)

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 Approximation

MetricTravelingSalespersonProblem: TSP where distances obey triangle inequality

Metric TSP – MST Approximation

METRICTRAVELINGSALESPERSONPROBLEM: TSP where distances obey triangle inequality

Step 1: MST

- Consider edge-weighted complete graph $G = ([n], E, D)$ of cities with pairwise distances $D_{i,j}$.
- Compute a minimum spanning tree T in G.

Metric TSP – MST Approximation

MetricTravelingSalespersonProblem: TSP where distances obey triangle inequality

Step 1: MST

- Consider edge-weighted complete graph $G = ([n], E, D)$ of cities with pairwise distances $D_{i,j}$.
- Compute a minimum spanning tree T in G.

"Baby-Christofides": Walk around T (Euler tour after doubling all edges)
If this visits a vertex another time, simply skip it
(shortcut edge to next vertex)

Lemma 10.20

Baby-Christofides is a 2-approximation for MetricTSP.

Metric TSP – MST Approximation

MetricTravelingSalespersonProblem: TSP where distances obey triangle inequality

Step 1: MST

- Consider edge-weighted complete graph $G = ([n], E, D)$ of cities with pairwise distances $D_{i,j}$.
- Compute a minimum spanning tree T in G.

"Baby-Christofides": Walk around T (Euler tour after doubling all edges)
If this visits a vertex another time, simply skip it
(shortcut edge to next vertex)

Lemma 10.20

Baby-Christofides is a 2-approximation for MetricTSP.

Proof:

⇝ Walking around T uses each edge twice: cost $= 2c(T)$.

Metric TSP – MST Approximation

MetricTravelingSalespersonProblem: TSP where distances obey triangle inequality

Step 1: MST

- Consider edge-weighted complete graph $G = ([n], E, D)$ of cities with pairwise distances $D_{i,j}$.
- Compute a minimum spanning tree T in G.

"Baby-Christofides": Walk around T (Euler tour after doubling all edges)
If this visits a vertex another time, simply skip it
(shortcut edge to next vertex)

Lemma 10.20

Baby-Christofides is a 2-approximation for MetricTSP.

Proof:

$\rightsquigarrow$ Walking around T uses each edge twice: cost $= 2c(T)$.

- Shortcutting does not make the tour longer by the triangle inequality.

Metric TSP – MST Approximation

MetricTravelingSalespersonProblem: TSP where distances obey triangle inequality

Step 1: MST

- Consider edge-weighted complete graph $G = ([n], E, D)$ of cities with pairwise distances $D_{i,j}$.
- Compute a minimum spanning tree T in G.

"Baby-Christofides": Walk around T (Euler tour after doubling all edges)
If this visits a vertex another time, simply skip it
(shortcut edge to next vertex)

Lemma 10.20

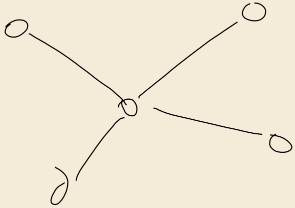
Baby-Christofides is a 2-approximation for MetricTSP.

Proof:

- $\rightsquigarrow$ Walking around T uses each edge twice: cost $= 2c(T)$.
- Shortcutting does not make the tour longer by the triangle inequality.
- Removing one edge form an optimal TSP tour yields a spanning tree (path)
- $\rightsquigarrow$ $OPT \geq c(T)$.

Matchings and Tours

Can we improve upon the specific Euler tour we used?



Matchings and Tours

Can we improve upon the specific Euler tour we used?

Doubling edges was costly. For even-degree vertices this is not needed!

Recall: graph has an Euler tour iff all vertices have even degrees.

Matchings and Tours

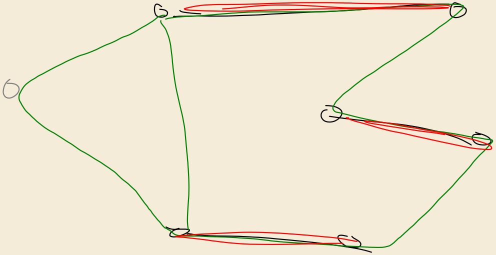
Can we improve upon the specific Euler tour we used?

Doubling edges was costly. For even-degree vertices this is not needed!

Recall: graph has an Euler tour iff all vertices have even degrees.

Lemma 10.21

Let $V' \subseteq V$ with $|V'|$ even and let M be a minimum-cost perfect matching on V' (in the TSP graph). Then $c(M) \leq OPT/2$.



Matchings and Tours

Can we improve upon the specific Euler tour we used?

Doubling edges was costly. For even-degree vertices this is not needed!

Recall: graph has an Euler tour iff all vertices have even degrees.

Lemma 10.21
Let $V' \subseteq V$ with $|V'|$ even and let M be a minimum-cost perfect matching on V' (in the TSP graph). Then $c(M) \leq OPT/2$.

Proof:
Let C^* be the optimal TSP tour and let C' be the your (on V') where we shortcut all vertices not in V'.

Matchings and Tours

Can we improve upon the specific Euler tour we used?

Doubling edges was costly. For even-degree vertices this is not needed!

Recall: graph has an Euler tour iff all vertices have even degrees.

Lemma 10.21

Let $V' \subseteq V$ with $|V'|$ even and let M be a minimum-cost perfect matching on V' (in the TSP graph). Then $c(M) \leq OPT/2$.

Proof:

Let C^* be the optimal TSP tour and let C' be the your (on V') where we shortcut all vertices not in V'.

By triangle inequality $c(C') \leq c(C^*)$.

Matchings and Tours

Can we improve upon the specific Euler tour we used?

Doubling edges was costly. For even-degree vertices this is not needed!

Recall: graph has an Euler tour iff all vertices have even degrees.

Lemma 10.21

Let $V' \subseteq V$ with $|V'|$ even and let M be a minimum-cost perfect matching on V' (in the TSP graph). Then $c(M) \leq OPT/2$.

Proof:

Let C^* be the optimal TSP tour and let C' be the your (on V') where we shortcut all vertices not in V'.
By triangle inequality $c(C') \leq c(C^*)$.

Since $|V'|$ is even, C' is the *disjoint union of two perfect matchings* of V' (odd and even steps).

Matchings and Tours

Can we improve upon the specific Euler tour we used?

Doubling edges was costly. For even-degree vertices this is not needed!

Recall: graph has an Euler tour iff all vertices have even degrees.

Lemma 10.21

Let $V' \subseteq V$ with $|V'|$ even and let M be a minimum-cost perfect matching on V' (in the TSP graph). Then $c(M) \leq OPT/2$. ◀

Proof:

Let C^* be the optimal TSP tour and let C' be the your (on V') where we shortcut all vertices not in V'.

By triangle inequality $c(C') \leq c(C^*)$.

Since $|V'|$ is even, C' is the *disjoint union of two perfect matchings* of V' (odd and even steps).

So the cheaper of these two matchings has cost $\leq c(C')/2 \leq c(C^*)/2 = OPT/2$.

Matchings and Tours

Can we improve upon the specific Euler tour we used?

Doubling edges was costly. For even-degree vertices this is not needed!

Recall: graph has an Euler tour iff all vertices have even degrees.

Lemma 10.21

Let $V' \subseteq V$ with $|V'|$ even and let M be a minimum-cost perfect matching on V' (in the TSP graph). Then $c(M) \le OPT/2$. ◀

Proof:

Let C^* be the optimal TSP tour and let C' be the your (on V') where we shortcut all vertices not in V'.
By triangle inequality $c(C') \le c(C^*)$.

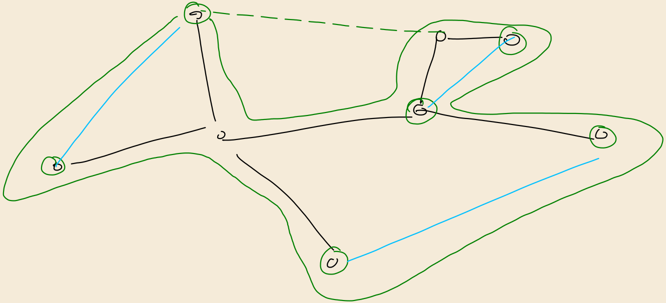
Since $|V'|$ is even, C' is the *disjoint union of two perfect matchings* of V' (odd and even steps).

So the cheaper of these two matchings has cost $\le c(C')/2 \le c(C^*)/2 = OPT/2$.
$\rightsquigarrow$ optimal perfect matching also has $c(M) \le OPT/2$. ■

Christofides's 3/2-Approximation

Step 2: Christofides's Algorithm

- T := MST in G.
- V' := vertices with odd degree in T.
- M := minimum-cost perfect matching of V' in G.
- Output Euler cycle C in $([n], E(T) \cup M)$, shortcutting repeated vertices.



Christofides's 3/2-Approximation

Step 2: Christofides's Algorithm

- T := MST in G.
- V' := vertices with odd degree in T.
- M := minimum-cost perfect matching of V' in G.
- Output Euler cycle C in $([n], E(T) \cup M)$, shortcutting repeated vertices.

Theorem 10.22

Christofides's algorithms is a $\frac{3}{2}$-approximation for MetricTSP. ◀

Christofides's 3/2-Approximation

Step 2: Christofides's Algorithm

- T := MST in G.
- V' := vertices with odd degree in T.
- M := minimum-cost perfect matching of V' in G.
- Output Euler cycle C in $([n], E(T) \cup M)$, shortcutting repeated vertices.

Theorem 10.22

Christofides's algorithms is a $\frac{3}{2}$-approximation for MetricTSP. ◀

Proof:

$$c(C) \quad = \quad c(T) + c(M) \quad \leq \quad OPT + OPT/2 \quad = \quad \frac{3}{2} \cdot OPT \qquad \blacksquare$$

Christofides's 3/2-Approximation

Step 2: Christofides's Algorithm

- T := MST in G.
- V' := vertices with odd degree in T.
- M := minimum-cost perfect matching of V' in G.
- Output Euler cycle C in $([n], E(T) \cup M)$, shortcutting repeated vertices.

Theorem 10.22

Christofides's algorithms is a $\frac{3}{2}$-approximation for METRICTSP.

Proof:

$$c(C) \;=\; c(T) + c(M) \;\leq\; OPT + OPT/2 \;=\; \frac{3}{2} \cdot OPT \qquad \blacksquare$$

Major open problem: Can $\frac{3}{2}$ be improved?

- Was open since 1976
- (Tiny) improvement published at STOC 2021 ($(\frac{3}{2} - \delta)$-approximation) out of PhD project of Nathan Klein (!)

10.8 Randomized Approximations

Randomized Approximation Guarantees

Definition 10.23 (Randomized δ-approx.)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, \mathbf{min})$ 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 10.24 (δ-expected approx.)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, \min)$ an optimization problem.
For $\delta > 1$, a 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.)