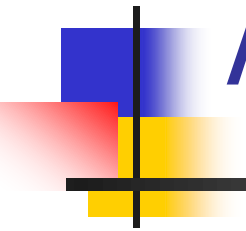


Chapter 35
Approximation Algorithm

Introduction to Data Structures

Kyuseok Shim

ECE, SNU.

Outline

- In this chapter, we present some examples of polynomial-time approximation algorithms for several NP-complete problems.

Coping with NP-Complete Problems

- We have at least three ways to get around NP-completeness.
  - If the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
  - We may be able to isolate important special cases that we can solve in polynomial time.
  - We might come up with approaches to find near-optimal solutions in polynomial time (either in the worst case or the expected case).
- We call an algorithm that returns near-optimal solutions an approximation algorithm.

Approximation Ratios

- Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution.
- Depending on the problem, we may define an optimal solution as one with maximum possible cost or one with minimum possible cost.
- Note that we assume that all solutions have positive cost, these ratios are always well defined.

Approximation Ratios

- We say that an algorithm for a problem has an approximation ratio $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C* of an optimal solution:
  - $\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$
  - For maximization problem, $1 \leq \frac{C^*}{C} \leq \rho(n)$
  - For minimization problem, $1 \leq \frac{C}{C^*} \leq \rho(n)$
- If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-approximation algorithm.

Approximation Ratios

- Because we assume that all solutions have positive cost, the approximation ratios are always well defined.
- The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* \leq 1$ implies $C^*/C \geq 1$.
- Therefore, a 1-approximation algorithm produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

Approximation Ratios

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

Approximation Scheme

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

Approximation Scheme

- Polynomial-time approximation scheme
  - For any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.
  - The running time can increase very rapidly as ϵ decreases.
  - e.g.) a scheme with $O(n^{2/\epsilon})$ time
- Ideally, if ϵ decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor.

Approximation Scheme

- Fully polynomial-time approximation scheme
  - Running time is polynomial both in $1/\epsilon$ and in the size n of the input instance.
  - With such a scheme, any constant-factor decrease in ϵ comes with a corresponding constant-factor increase in the running time.
  - e.g.) a scheme with $O((1/\epsilon)^2 n^3)$ time

VERTEX-COVER Problem

VERTEX-COVER Problem

- Vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$.
- Vertex cover problem is to find an optimal vertex cover, or a vertex cover of minimum size in a given undirected graph.
- This problem is the optimization version of an NP-complete decision problem.
- Even though we don't know how to find an optimal vertex cover in a graph G in polynomial time, we can efficiently find a vertex cover that is near-optimal.

Approximation Algorithm for VERTEX-COVER Problem

- The following algorithm returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.
- Its running time is O (|V| + |E|) time using adjacency lists to represent E′.

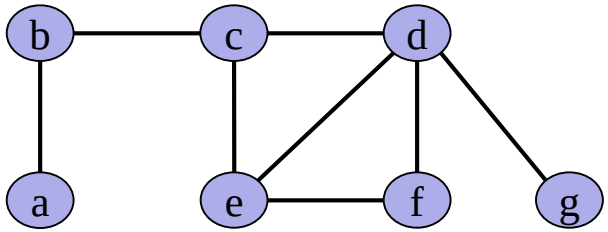
APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

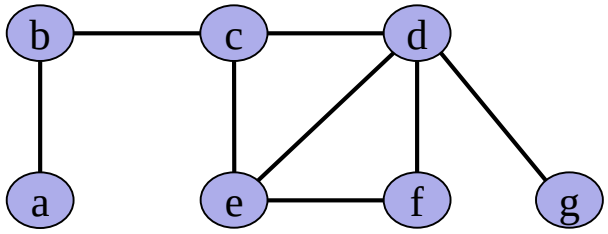


Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C=∅



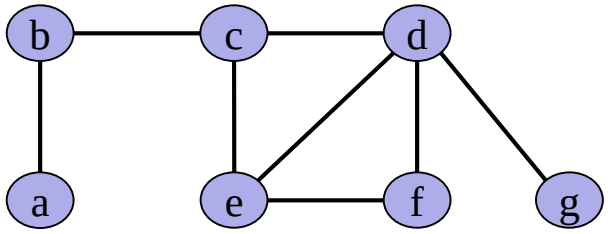
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C=∅

E′={(a,b), (b,c), (c,d), (c,e), (d,e), (d,f), (e,f), (d,g)}



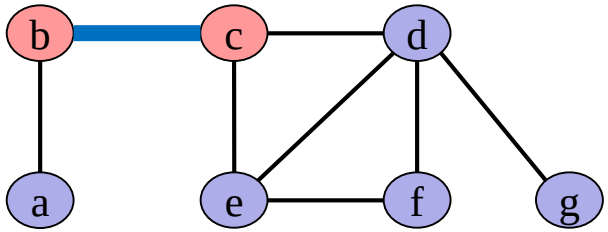
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C=∅

E′={(a,b), (b,c), (c,d), (c,e), (d,e), (d,f), (e,f), (d,g)}



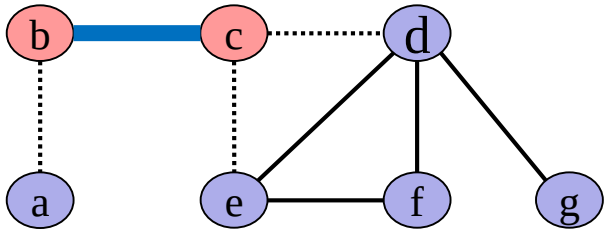
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C={b,c}

E′={(d,e), (d,f), (e,f), (d,g)}



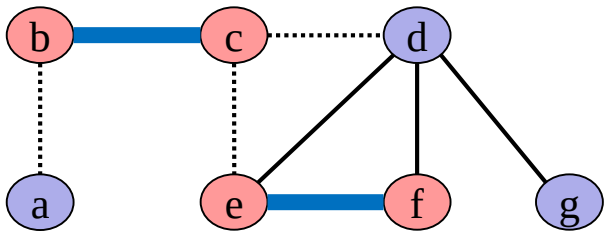
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C={b,c}

E′={(d,e), (d,f), (e,f), (d,g)}



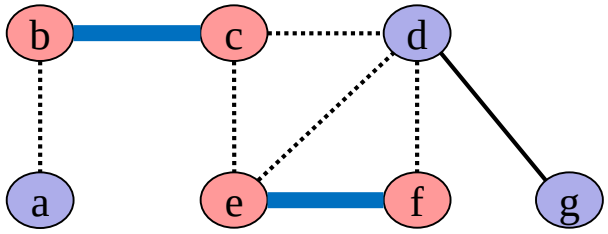
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C={b,c,e,f}

E′={(d,g)}



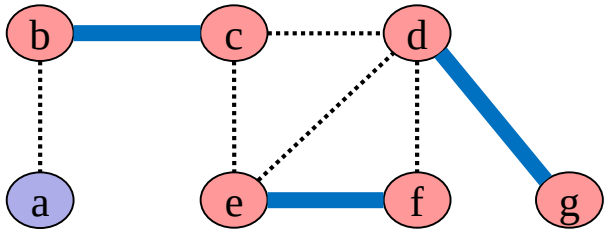
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C={b,c,e,f}

E′={(d,g)}



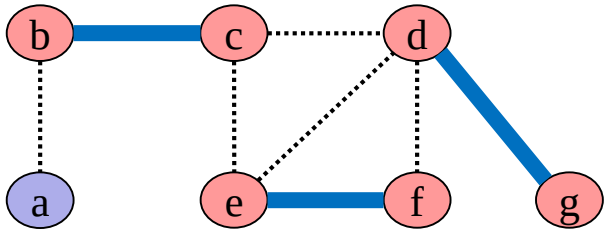
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C={b,c,e,f,d,g}

E′=∅



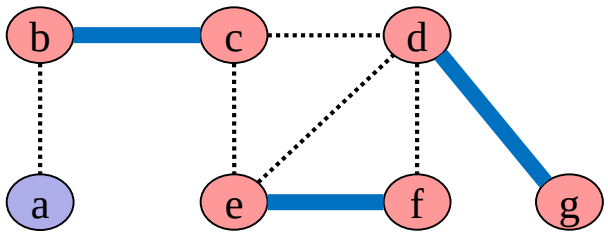
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

C={b,c,e,f,d,g}

E′=∅



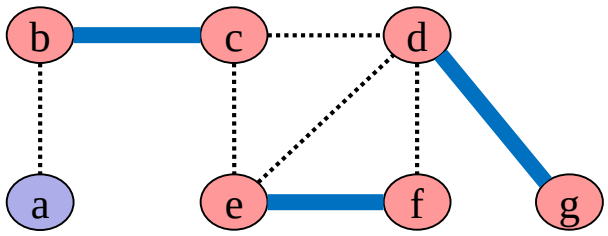
Approximation Algorithm for VERTEX-COVER Problem

APPROX-VERTEX-COVER(G)

1. C = Ø
2. E′ = G.E
3. **while** E′ ≠ Ø
4. let (u, v) be an arbitrary edge of E ′
5. C = C ∪ {u, v }
6. remove from E′ every edge incident on either u or v
7. **return** C

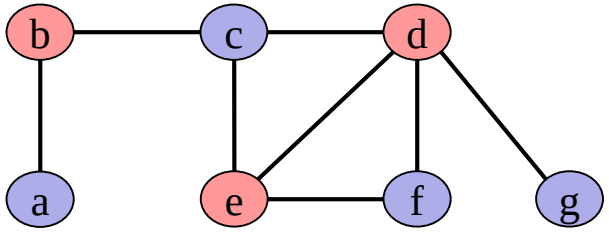
C={b,c,e,f,d,g}

E′=∅



Approximation Algorithm for VERTEX-COVER Problem

- An optimal solution



Approximation Algorithm for VERTEX-COVER Problem

- Theorem 35.1
  - APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.
- Proof
  - We have already shown that APPROX-VERTEX-COVER runs in polynomial time.
  - The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in G.E has been covered by some vertex in C.
  - Let A represent the set of edges (and their endpoints) that are included in C.

Approximation Algorithm for VERTEX-COVER Problem

- Theorem 35.1
  - APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.
- Proof
  - In order to cover the edges in A, an optimal vertex cover C^* must include at least one endpoint of each edge that is included in C.
  - Since once an edge is picked, all other edges that are incident on its endpoints are deleted from E' and thus no two edges in A share an endpoint.
  - Thus, no two edges in A are covered by the same vertex in C^* and we have $|C^*| \geq |A|$.
  - Each iteration of APPROX-VERTEX-COVER picks an edge for which neither endpoint is in C

    ➔ $|C| = 2|A|$ and $|C| \leq 2|C^*|$

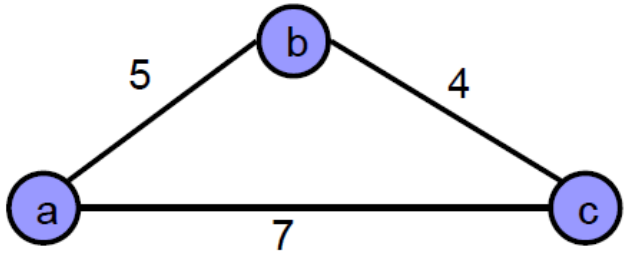
TRAVELING-SALESMAN Problem

TRAVELING-SALESMAN Problem

- In the Traveling Salesman problem, we are given an undirected complete graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge (u, v).
- Traveling Salesman problem is to find a hamiltonian cycle, or a tour of G with minimum cost.
- This problem is the optimization version of an NP-complete decision problem.
- Even though we don't know how to find a tour in a graph G in polynomial time, we can efficiently find a tour that is near-optimal.

Approximation Algorithm for TRAVELING-SALESMAN Problem

- Triangle inequality
  - In many practical situations, the least costly way to go from a place u to a place w is to go directly
  - For all vertices $u, v, w \in V$, we have $c(u, w) \leq c(u, v) + c(v, w)$



Approximation Algorithm for TRAVELING-SALESMAN Problem

- The following algorithm returns tour whose cost is guaranteed to be no more than twice the cost of an optimal tour.

APPROX-TSP-TOUR(G, c)

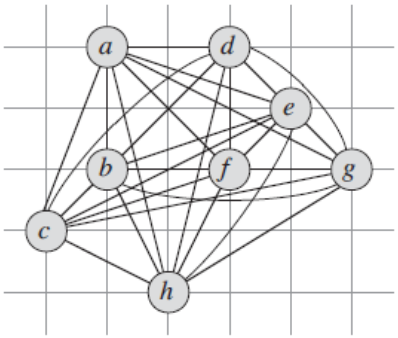
1. select a vertex $r \in G.V$ to be a "root" vertex $O(1)$
2. compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r) $O(|V|^2)$
3. let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T $O(|V|)$
4. **return** the hamiltonian cycle H

- Its running time is $O(|V|^2)$ time.

Approximation Algorithm for TRAVELING-SALESMAN Problem

APPROX-TSP-TOUR(G, c)

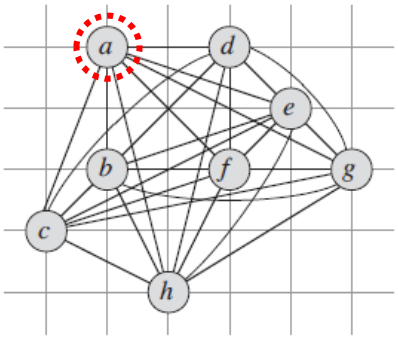
1. select a vertex $r \in G.V$ to be a "root" vertex
2. compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T
4. **return** the hamiltonian cycle H



Approximation Algorithm for TRAVELING-SALESMAN Problem

APPROX-TSP-TOUR(G, c)

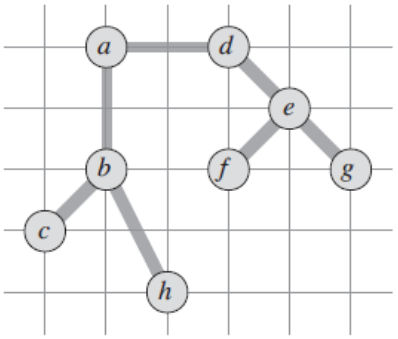
1. select a vertex $r \in G.V$ to be a "root" vertex
2. compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T
4. **return** the hamiltonian cycle H



Approximation Algorithm for TRAVELING-SALESMAN Problem

APPROX-TSP-TOUR(G, c)

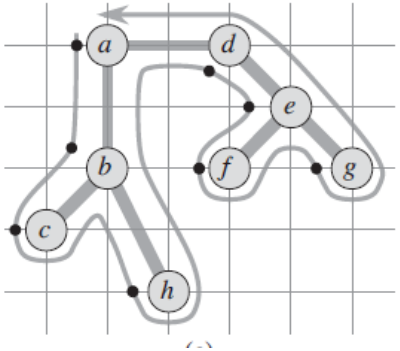
1. select a vertex $r \in G.V$ to be a "root" vertex
2. compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T
4. **return** the hamiltonian cycle H



Approximation Algorithm for TRAVELING-SALESMAN Problem

APPROX-TSP-TOUR(G, c)

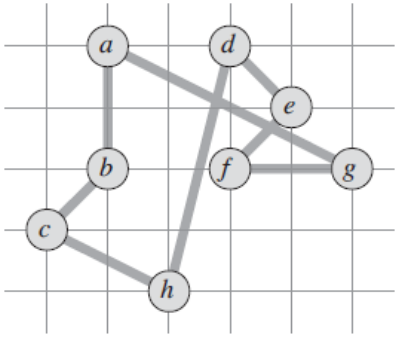
1. select a vertex $r \in G.V$ to be a "root" vertex
2. compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T
4. **return** the hamiltonian cycle H



Approximation Algorithm for TRAVELING-SALESMAN Problem

APPROX-TSP-TOUR(G, c)

1. select a vertex $r \in G.V$ to be a "root" vertex
2. compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T
4. **return** the hamiltonian cycle H

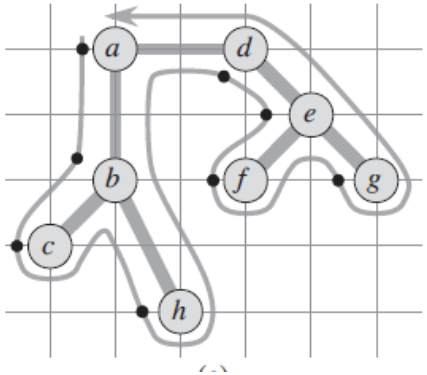


Approximation Algorithm for TRAVELING-SALESMAN Problem

- Theorem 35.2
  - APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the TRAVELING-SALESMAN problem with the triangle inequality.
- Proof
  - We have already shown that APPROX-TSP-TOUR runs in polynomial time.
  - Let H^* denote an optimal tour.
  - We obtain a spanning tree T' by deleting any edge from the tour H^*, and each edge is nonnegative.
  - Let T be a minimum spanning tree of G.
  - Thus, the we have $c(T) \leq c(T') \leq c(H^*)$ (35.4)

Approximation Algorithm for TRAVELING-SALESMAN Problem

- Proof (contd.)
  - Let W be a full walk of the minimum spanning tree T.
    - A full walk lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree.

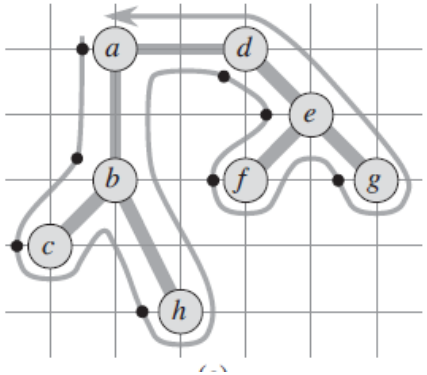


Full walk

$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$

Approximation Algorithm for TRAVELING-SALESMAN Problem

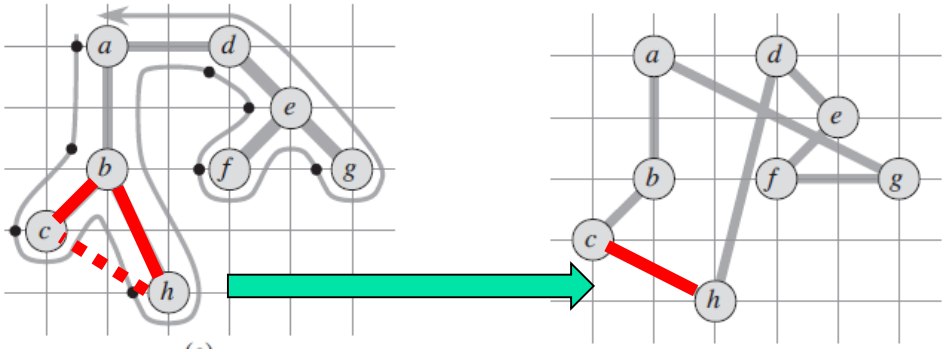
- Proof (cont.)
  - Since the full walk traverses every edge of T exactly twice, we have
    - $c(W) = 2c(T)$ (35.5)
  - Since we have $c(T) \leq c(H^*)$ (35.4), we obtain
    $$c(W) \leq 2c(H^*) \quad (35.6)$$



Full walk
$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$

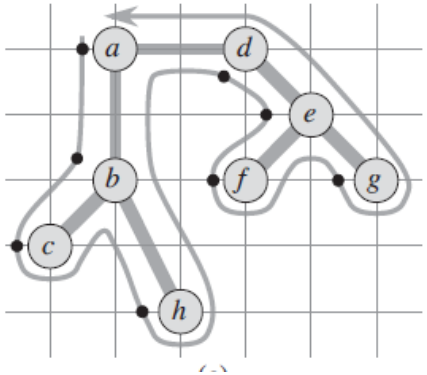
Approximation Algorithm for TRAVELING-SALESMAN Problem

- Proof (cont.)
  - By the triangle inequality, we can delete a visit to any vertex from W and the cost does not increase.
    - If we delete a vertex v from W between visits to u and w, the resulting ordering specifies going directly from u to w.



Approximation Algorithm for TRAVELING-SALESMAN Problem

- Proof (cont.)
  - By repeatedly applying this operation, we can remove from W all but the first visit to each vertex.
  - This ordering is the same as that obtained by a preorder walk of the tree T.



deleted

Full walk

$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$

Preorder walk

a, b, c, h, d, e, f, g

Approximation Algorithm for TRAVELING-SALESMAN Problem

- Proof (cont.)
  - Let H be the cycle corresponding to this preorder walk.
    - It is a hamiltonian cycle, since every vertex is visited exactly once.
    - It is the cycle computed by APPROX-TSP-TOUR.
  - Since H is obtained by deleting vertices from the full walk W and
    - $c(T) \leq c(T') \leq c(H^*)$ (35.4)
    - $c(W) = 2c(T) \leq 2c(H^*)$ (35.5) and (35.6)
  - Thus, we have
    - $c(H) \leq c(W) = 2c(T) \leq 2c(H^*)$

The General TRAVELING-SALESMAN Problem

- If we drop the assumption that the cost function c satisfies the triangle inequality, then we cannot find good approximate tours in polynomial time unless $P = NP$.
- Theorem 35.3
  - If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesman problem.

The General TRAVELING-SALESMAN Problem

- Proof sketch (by contradiction)
  - Suppose to the contrary that for some number $\rho \geq 1$, there is a polynomial-time approximation algorithm A with approximation ratio ρ.
  - Without loss of generality, we assume that ρ is an integer, by rounding it up if necessary.
  - We shall then show how to use A to solve instances of the HAMILTONIAN-CYCLE problem in polynomial time.
  - Since the HAMILTONIAN-CYCLE problem is NP-complete, if we can solve it in polynomial time, then $P = NP$.

The General TRAVELING-SALESMAN Problem

- Proof
  - Let $G = (V, E)$ be an instance of the HAMILTONIAN-CYCLE problem.
  - We wish to determine efficiently whether G contains a hamiltonian cycle by making use of the hypothesized approximation algorithm A.

The General TRAVELING-SALESMAN Problem

- Proof (cont.)
  - We turn G into an instance of the TRAVELING-SALESMAN problem in $O(|V|^2)$ time as follows.
    - Let $G' = (V, E')$ be the complete graph on V.
    - Assign an integer cost to each edge in E' as follows
      - $c(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E, \\ \rho|V| + 1 & \text{otherwise} \end{cases}$

The General TRAVELING-SALESMAN Problem

- Proof (cont.)
  - Consider the TRAVELING-SALESMAN problem with G' and the cost function c.
  - If the original graph G has a hamiltonian cycle H, the cost function c assigns to each edge of H a cost of 1, and so G' contains a tour of cost $|V|$.
  - On the other hand, if G does not contain a hamiltonian cycle, then any tour of G' must use some edge not in E.
  - But any tour that uses an edge not in E has a cost of at least
    - $(\rho|V| + 1) + (|V| - 1) = \rho|V| + |V| > \rho|V|$

The General TRAVELING-SALESMAN Problem

- Proof (cont.)
  - Because edges not in G are so costly, there is a gap of at least $\rho|V|$ between the cost of a tour that is a hamiltonian cycle in G (cost $|V|$) and the cost of any other tour (cost at least $\rho|V| + |V|$).
  - Thus, the cost of a tour that is not a hamiltonian cycle in G is at least a factor of $\rho + 1$ greater than the cost of a tour that is a hamiltonian cycle in G.

The General TRAVELING-SALESMAN Problem

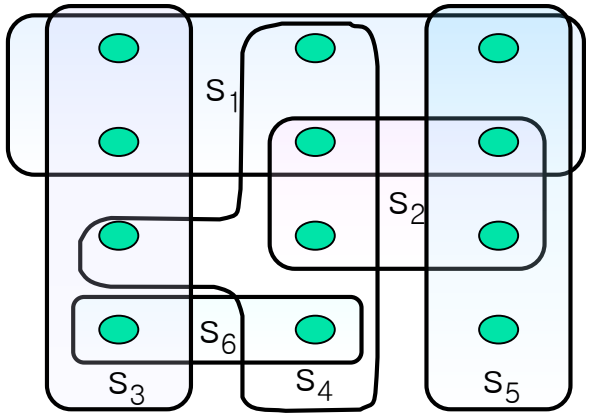
- Proof (cont.)
  - Suppose that we apply a ρ-approximation algorithm A to the TRAVELING-SALESMAN problem with G' and c.
  - If G contains a hamiltonian cycle, then A must return a tour of cost less than $\rho|V|$.
    - G' has an optimal tour with cost $|V|$ only.
  - If G has no hamiltonian cycle, then A returns a tour of cost more than $\rho|V|$.
    - G' has an optimal tour with cost at least $(\rho + 1)|V|$.
  - Thus, we can use A to solve the HAMILTONIAN-CYCLE problem in polynomial time.

SET-COVERING Problem

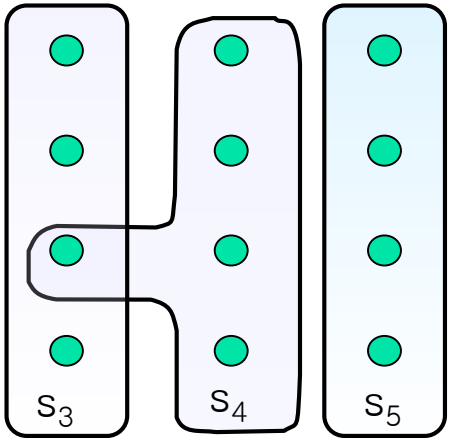
SET-COVERING Problem

- An optimization problem that models many problems that require resources to be allocated.
- Its corresponding decision problem generalizes the NP-complete vertex cover problem and is therefore also NP-hard.
- The approximation algorithm developed to handle the vertex-cover problem doesn't apply here, however, and so we need to try other approaches.
- We shall examine a simple greedy heuristic with a logarithmic approximation ratio.

An Example of the Set-Covering Problem



An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 points and $\mathcal{F}$ ={S_1, S_2, S_3, S_4, S_5, S_6}



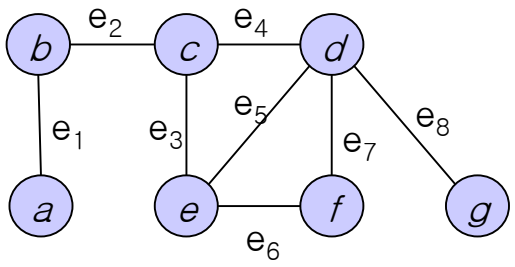
A minimum size set cover is {S_3, S_4, S_5} with size 3.

Problem Formulation

- Given an instance $(X, \mathcal{F})$ of the set-covering problem where
  - a finite set X
  - a family $\mathcal{F}$ of subsets of X
  - $X = \bigcup_{S \in \mathcal{F}} S$
  - We say a subset $S \in \mathcal{F}$ covers its elements.
- Find a minimum-size subset $\mathcal{C} \subset \mathcal{F}$ whose members cover all of X ($\mathcal{C}$: set-cover)

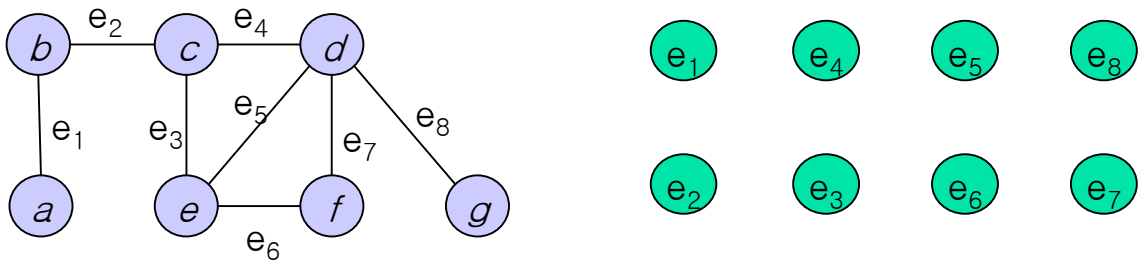
NP-Completeness of the Set-Covering Problem

- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.



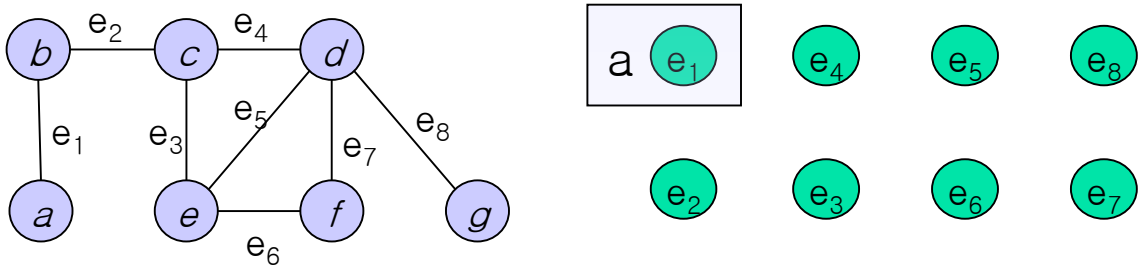
NP-Completeness of the Set-Covering Problem

- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.



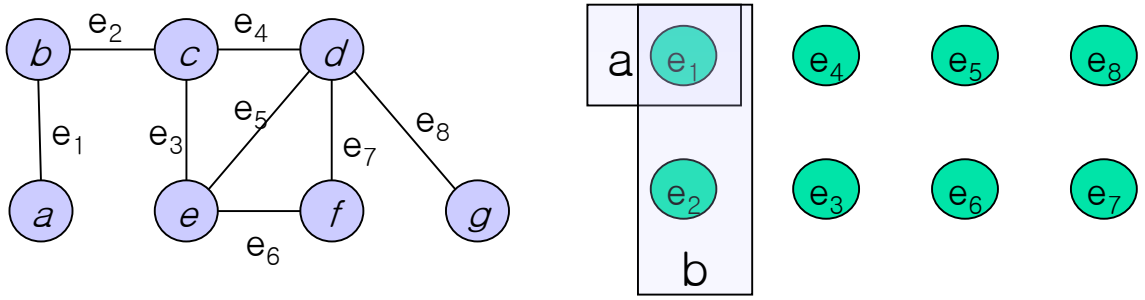
NP-Completeness of the Set-Covering Problem

- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.



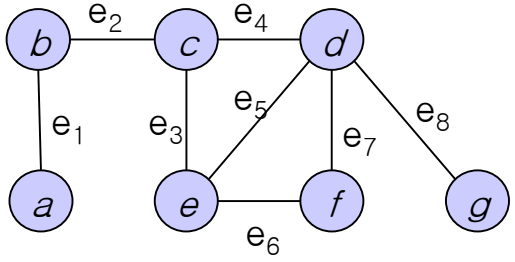
NP-Completeness of the Set-Covering Problem

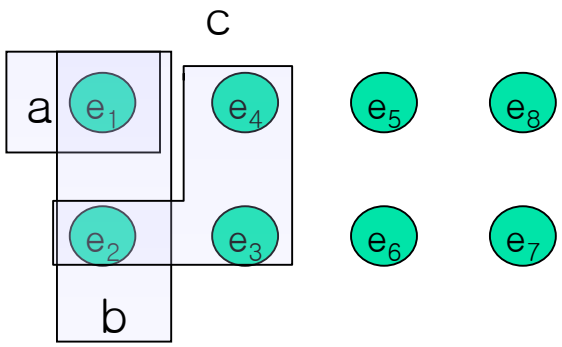
- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.



NP-Completeness of the Set-Covering Problem

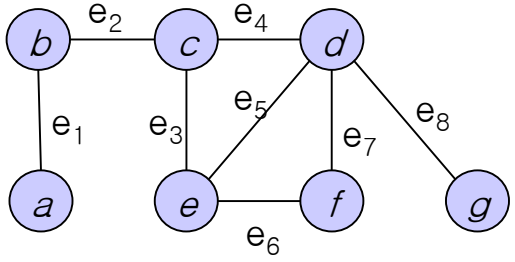
- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.

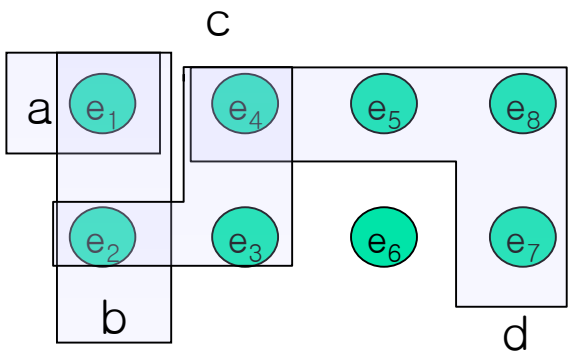




NP-Completeness of the Set-Covering Problem

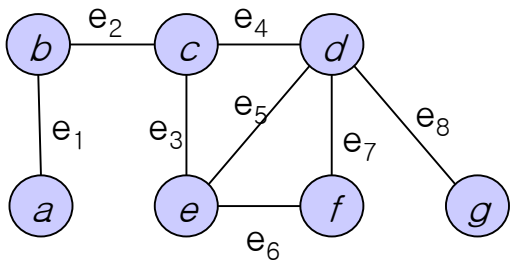
- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.

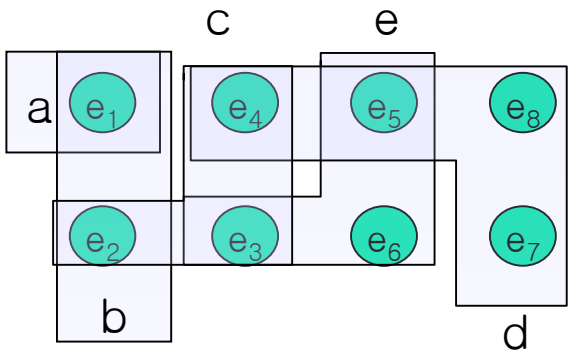




NP-Completeness of the Set-Covering Problem

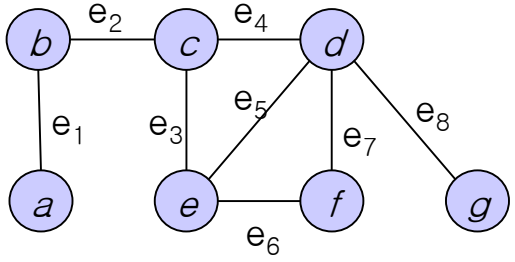
- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.

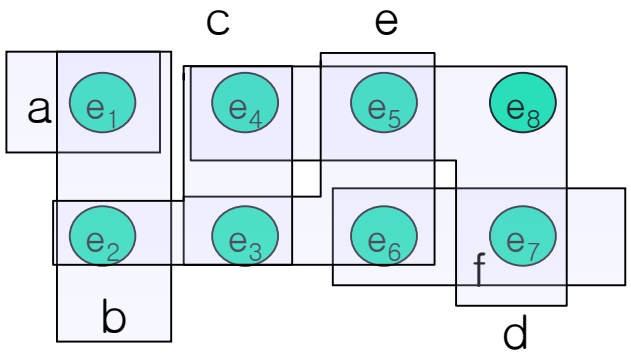




NP-Completeness of the Set-Covering Problem

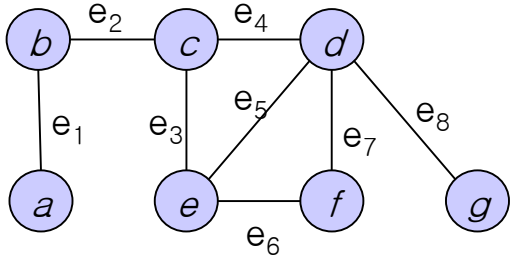
- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.

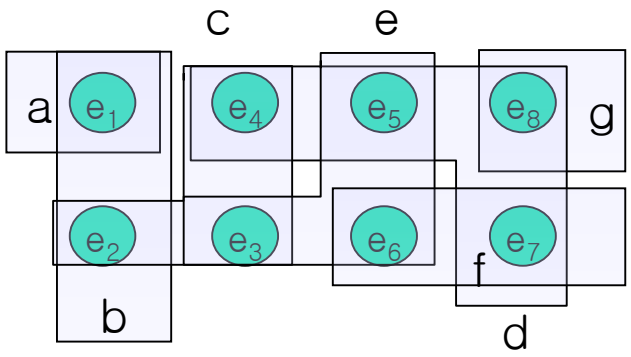




NP-Completeness of the Set-Covering Problem

- Given a directed graph G=(V,E) and integer k, define X=E and $\mathcal{F}$=V.
  - The reduction is done in polynomial time.



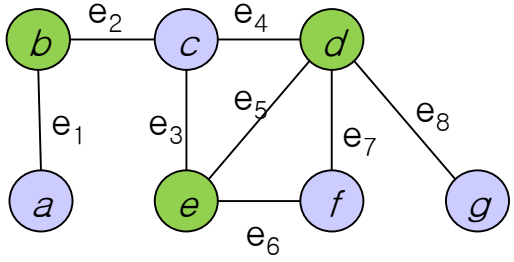


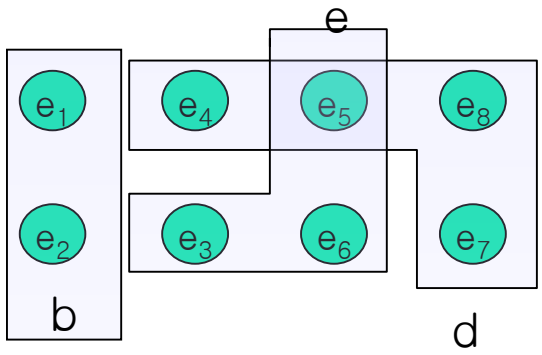
NP-Completeness of the Set-Covering Problem

- The set-covering problem is NP-hard.
  - The graph G has a vertex cover of size k if and only if the set X has a set-cover of size k.
- The set-covering problem is in NP.
  - Given a set-cover C, that |C|=k and C covers X can be verified in polynomial time.
- Thus, the set-covering problem is NP-complete.

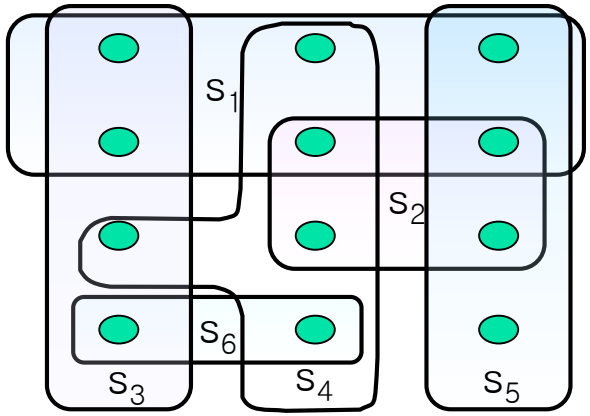
NP-Completeness of the Set-Covering Problem

- An optimal solution

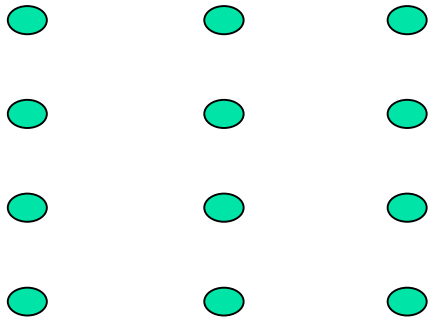




An Example of Greedy Approximation

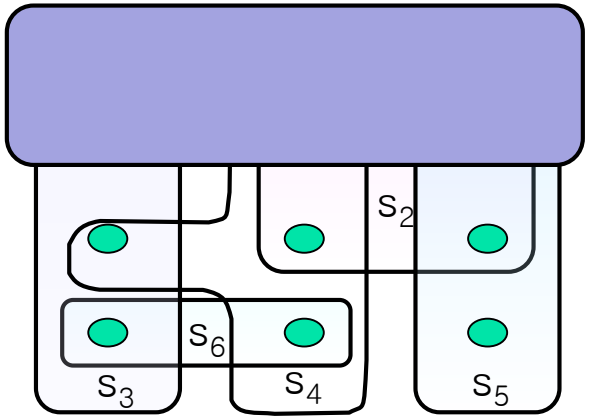


An instance of
the set covering problem

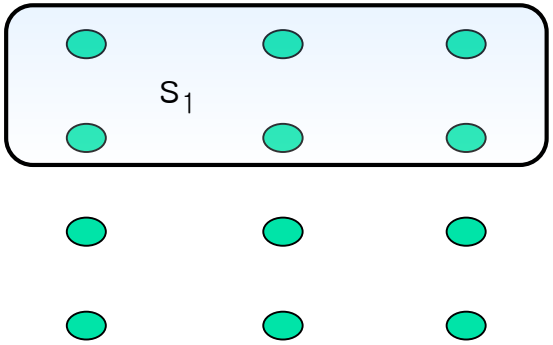


The greedy solution

An Example of Greedy Approximation



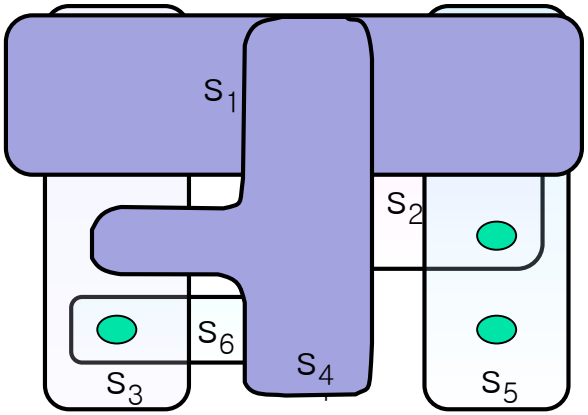
An instance of
the set covering problem



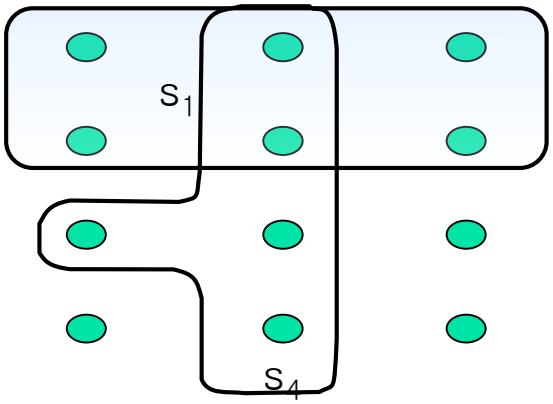
The greedy solution

S_1

An Example of Greedy Approximation

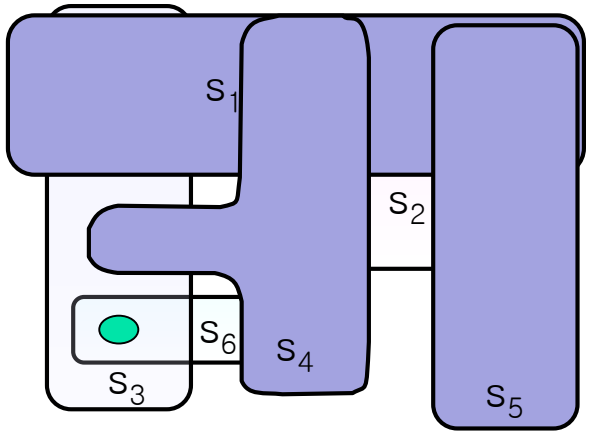


An instance of
the set covering problem

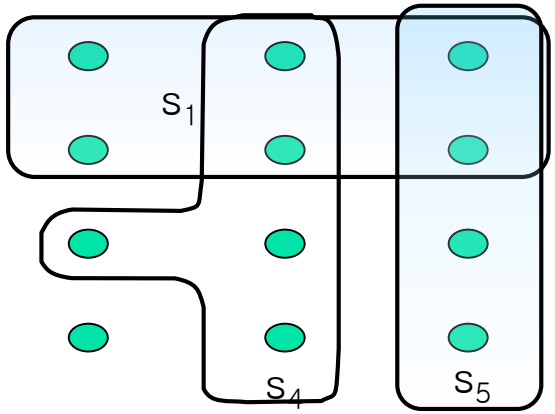


The greedy solution

An Example of Greedy Approximation

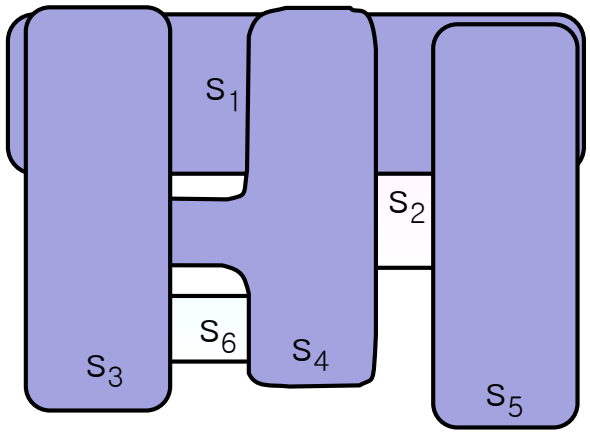


An instance of
the set covering problem

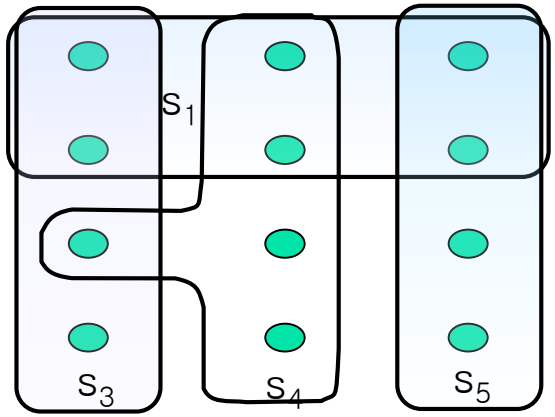


The greedy solution

An Example of Greedy Approximation



An instance of
the set covering problem



The greedy solution

A Greedy Approximation Algorithm

GREEDY-SET-COVER(X,$\mathcal{F}$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. **return** $\mathcal{C}$

A Greedy Approximation Algorithm

GREEDY-SET-COVER(X,$\mathcal{F}$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$ ← $\min(|X|, |\mathcal{F}|)$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. **return** $\mathcal{C}$

$O(|X||\mathcal{F}|)$

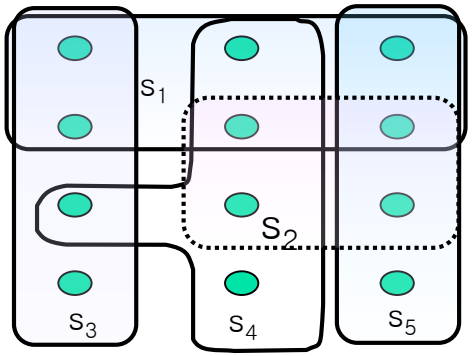
$O(|X||\mathcal{F}|\min(|X|, |\mathcal{F}|))$: time complexity

Proof of $\sum_{x \in S} c_x \leq H(|S|)$

- Consider any set $S \in \mathcal{F}$,
- Let $u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$ with i = 1,..., $|\mathcal{C}|$.
  - Note that u_i is the number of elements in S that remain uncovered after the algorithm has selected the sets S_1, S_2,..., S_i.
  - We define $u_0 = |S|$ to be the number of elements of S, which are all initially uncovered.
- Let k be the least index such that $u_k = 0$
  - Each element of S is covered by at least one of the sets S_1, S_2,... S_k.
  - Some elements in S is uncovered by $S_1 \cup S_2 \cup \cdots \cup S_{k-1}$.

An Example of u_i

- $u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$ where S can be the largest when S=X
- The greedy solution: S_1, S_4, S_5, S_3
  - $|S_1|=6$
  - $|S_4-S_1|=3$
  - $|S_5-(S_1 \cup S_4)|=2$
  - $|S_3-(S_1 \cup S_4 \cup S_5)|=1$
- Let $S=S_2$
  - $u_0=|S_2|=4$
  - $u_1=|S_2- S_1|=2$
  - $u_2=|S_2-(S_1 \cup S_4)|=1$
  - $u_3=|S_2-(S_1 \cup S_4 \cup S_5)|=0$
  - $\sum c_x = 2*1/6+1*1/3+1*1/2=4/3$



The greedy solution

Proof of $\sum_{x \in S} c_x \leq H(|S|)$

- Consider any set $S \in \mathcal{F}$.
- Let $u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$ for i = 1,…, $|\mathcal{C}|$.
- Let k be the least index such that $u_k = 0$
- Then, $u_{i-1} \geq u_i$, and $u_{i-1} - u_i$ elements of S are covered for the first time by S_i for i = 1,…,k.
- Thus, $\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$
- Since the greedy choice of S_i guarantees that S cannot cover more new elements than S_i does, observe that

$$|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|$$
$$= u_{i-1}$$

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

Proof of $\sum_{x \in S} c_x \leq H(|S|)$

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

  $\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j}$ (because $j \leq u_{i-1}$)

  $= \sum_{i=1}^{k} (\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j}) = \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i))$

  $= H(u_0) - H(u_k)$ (because the sum telescopes)

  $= H(u_0) - H(0)$

  $= H(u_0)$ (because $H(0) = 0$)

  $= H(|S|)$

A Greedy Approximation Algorithm

- Corollary 35.5
  - Greedy-Set-Cover is a polynomial-time ($\ln|X|+1$)-approximation algorithm.
- Proof
  - Use inequality $\sum_{k=1}^{n} \frac{1}{k} \leq \ln(n) + 1$ (A.14) and Theorem 35.4.

Subset-Sum Problem

Subset-Sum Problem

- Recall that an instance of the subset-sum problem is a pair (S, t), where
  - S is a positive integer set $\{x_1, x_2, \ldots x_n\}$
  - t is a positive integer
- This decision problem asks whether there exists a subset of S that adds up exactly to the target value t.
  - NP-Complete (see Section 34.5.5)
- e.g.)
  - S = {1, 2, 7, 14, 49, 54}, t = 58
  - The subset S′ = {2, 7, 49} is a solution.

Subset-Sum Problem

- The optimization problem associated with this decision problem arises in practical applications.
- We wish to find a subset of $\{x_1, x_2, \ldots x_n\}$ whose sum is as large as possible but not larger than t.
- For example, we may have a truck that can carry no more than t pounds, and n different boxes to ship, the i-th of which weighs x_i pounds.
- We wish to fill the truck with as heavy a load as possible without exceeding the weight limit.

Subset-Sum Problem

- We first present an exponential-time algorithm that computes the optimal value for this optimization problem.
- We next show how to modify the algorithm so that it becomes a fully polynomial-time approximation scheme.
- Recall that a fully polynomial-time approximation scheme has a running time that is polynomial in $1/\varepsilon$ as well as in the size of the input.

An Exponential-Time Exact Algorithm

- Preliminary
  - Given a integer set(list) S and a integer x
    - $S + x = \{s+x : s \in S\}$
    - e.g.) S={ 0, 1, 5, 9 }, S+2={ 2, 3, 7, 11 }
  - Let P_i denote the set of all possible summation values that can be obtained by selecting a subset of $\{ x_1, x_2, \cdots, x_i \}$
    - e.g.) S={1,4,5}, P_1={0,1}, P_2={0,1,4,5}, P_3={0,1,4,5,6,9,10}
    - $P_i = P_{i-1} \cup (P_{i-1} + x_i)$
  - Let L_i be a sorted list containing every element of P_i whose value is not more than t

An Exponential-Time Exact Algorithm

- Preliminary
  - We also use an auxiliary procedure MERGE-LISTS(L, L'), which returns the sorted list that is the merge of its two sorted input lists L and L' with duplicate values removed.
  - Like the MERGE procedure we used in merge sort, MERGE-LISTS runs in time $O(|L|+|L'|)$.
  - We omit the pseudocode for MERGE-LISTS.

An Exponential-Time Exact Algorithm

EXACT-SUBSET-SUM(S, t)

1. n = |S|
2. L_0 = <0>
3. **for** i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1}, $L_{i-1}+x_i$)
5. remove form L_i every element that is greater than t
6. **return** the largest element is L_n

- MERGE-LISTS(L, L')
  - Returns the sorted list like merge sort
  - O(|L|+|L'|)
  - Duplicate values are removed

An Exponential-Time Exact Algorithm

- EXACT-SUBSET-SUM is an exponential-time algorithm in general
  - Because $|L_i|$ can be as much as 2^i
- What is a special case where it becomes a polynomial-time algorithm?
  - When t is polynomial in |S| or
  - When all the numbers in S are bounded by a polynomial in |S|

Fully Polynomial-Time Approximation Scheme

- It is a $(1+\varepsilon)$-approximation algorithm.
- Running time is polynomial in $1/\varepsilon$ as well as in the size of the input.
  - e.g.) $O((1/\varepsilon)^2 n^3)$

Trimming

- Idea
  - If two values in L are close, since we want just an approximate solution, then there is no reason to maintain both explicitly.
- To trim a list L by δ (0<δ<1) means
  - To remove as many elements as possible, in a such way that, for every removed element y, there is an element z still in trimming result L' satisfying $\frac{y}{1+\delta} \leq z \leq y$.

Trimming

- For every removed element y, there is an element z still in trimming result L' satisfying

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

- e.g.)
  - δ = 0.1, L=<10,11,12,15,20,21,22,23,24,29>
  - L'=<10,12,15,20,23,29>
    - 11 is represented by 10 ($11/1.1 \leq 10 \leq 11$)
    - 21 and 22 are represented by 20
    - 24 is represented by 23

Trimming

- The following procedure trims list $L=\langle y_1, y_2, \cdots, y_m \rangle$ in time $\Theta(m)$, given L and δ, and assuming that L is sorted into monotonically increasing order.
- The output of the procedure is a trimmed, sorted list.

TRIM(L,δ)

1. let m be the length of L
2. L′ = $\langle y_1 \rangle$
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

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

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Trimming

- The procedure scans the elements of L in monotonically increasing order.
- A number is appended onto the returned list L′ only if it is the first element of L or if it cannot be represented by the most recent number placed into L′.

TRIM(L, δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

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

An Example of Trimming

TRIM(L,δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

$\delta = 0.1$

L	10	11	12	15	20	21	22	23	24	29

An Example of Trimming

TRIM(L,δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

$\delta = 0.1$

m = 10

L	10	11	12	15	20	21	22	23	24	29

An Example of Trimming

TRIM(L,δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

$\delta = 0.1$

m = 10

L	10	11	12	15	20	21	22	23	24	29

L′	10

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of Trimming

TRIM(L,δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

$\delta = 0.1$

m = 10

last = 10

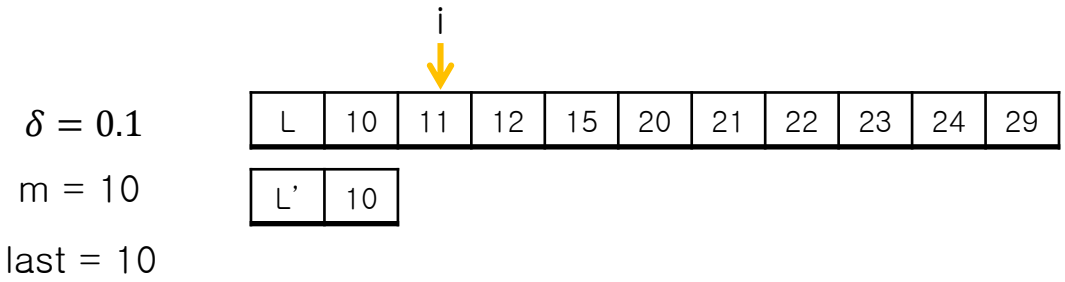
L	10	11	12	15	20	21	22	23	24	29

L′	10

An Example of Trimming

TRIM(L,δ)

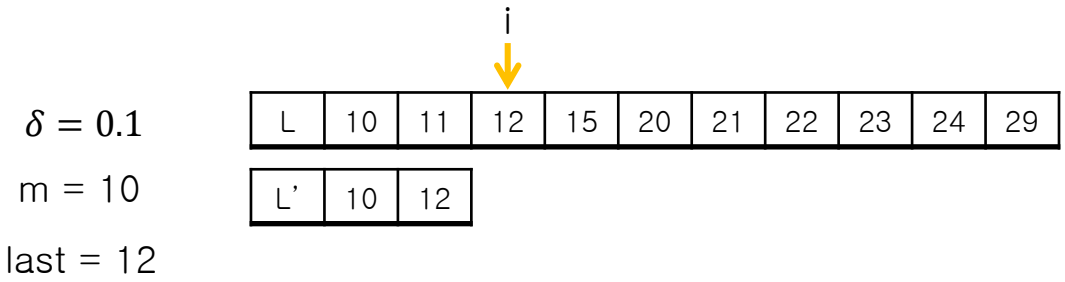
1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′



An Example of Trimming

TRIM(L,δ)

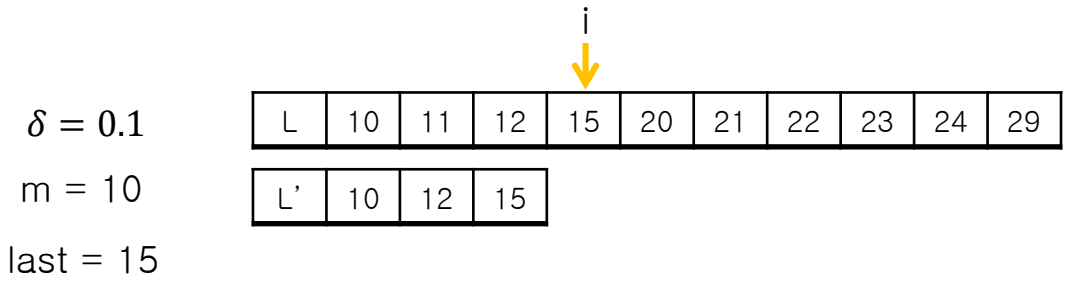
1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′



An Example of Trimming

TRIM(L,δ)

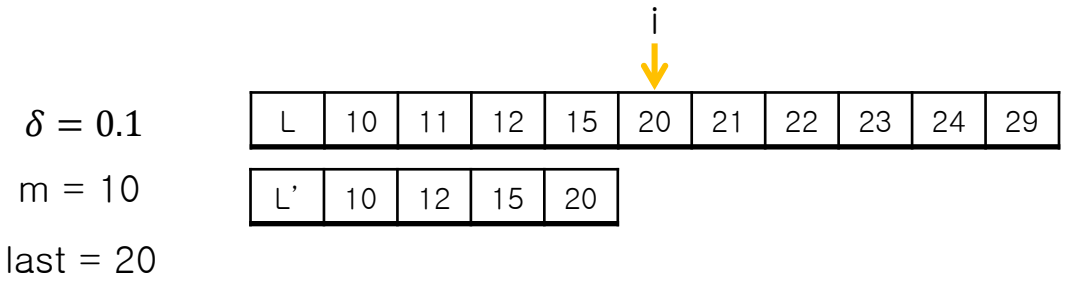
1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′



An Example of Trimming

TRIM(L,δ)

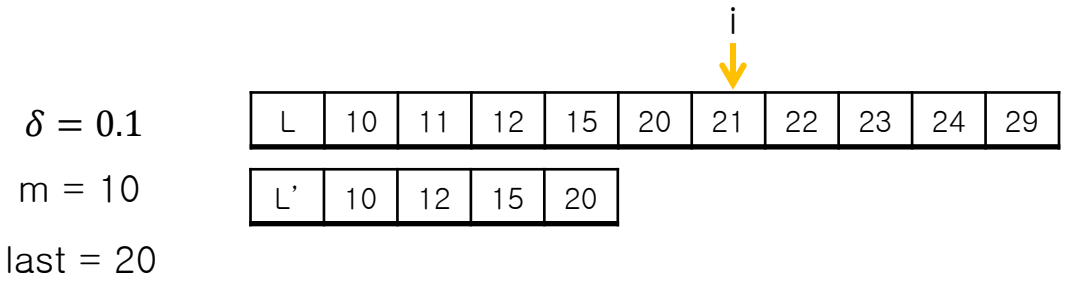
1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′



An Example of Trimming

TRIM(L,δ)

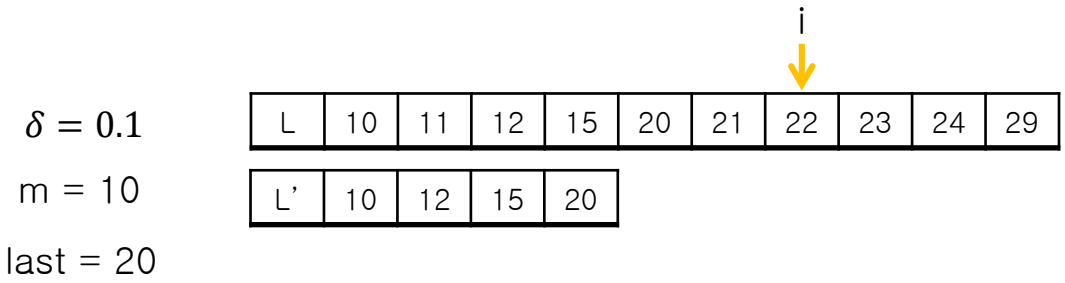
1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′



An Example of Trimming

TRIM(L,δ)

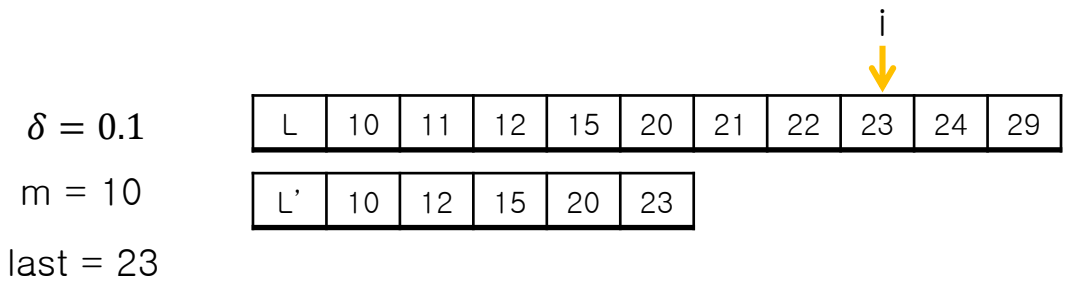
1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′



An Example of Trimming

TRIM(L,δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′



An Example of Trimming

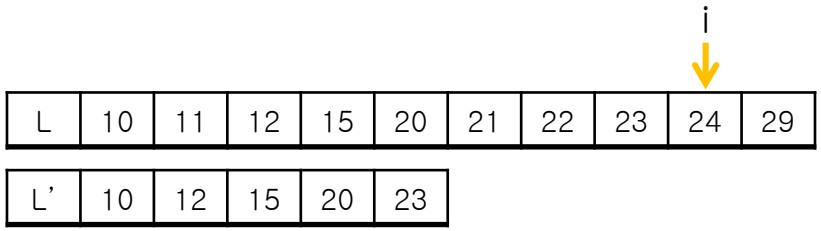
TRIM(L,δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

$\delta = 0.1$

m = 10

last = 23



An Example of Trimming

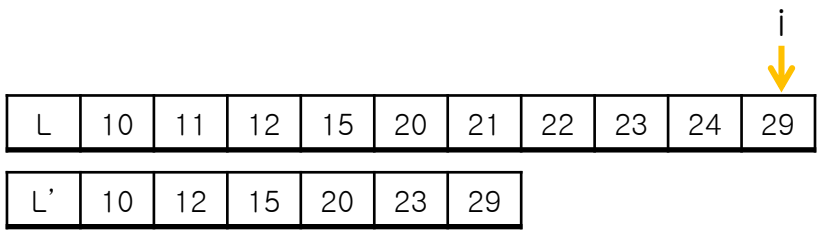
TRIM(L,δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

$\delta = 0.1$

m = 10

last = 29



An Example of Trimming

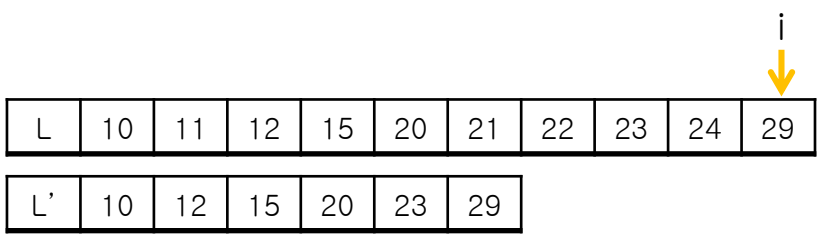
TRIM(L,δ)

1. let m be the length of L
2. L′ = <y_1>
3. last = y_1
4. **for** i =2 to m
5. **if** y_i>last(1+δ)
6. append y_i onto the end of L′
7. last = y_i
8. **return** L′

$\delta = 0.1$

m = 10

last = 29



APPROX-SUBSET-SUM

- Given input S, a target integer t, and an approximation parameter ε (0<ε<1),
- It returns a value z whose value is within a 1+ε factor of the optimal solution.

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

n = 4

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

n = 4

L_0	0

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_1	0	104

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_1	0	104

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_1	0	104

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_2	0	102	104	206

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_2	0	102	206

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_2	0	102	206

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_3	0	102	201	206	303	407

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_3	0	102	201	303	407

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_3	0	102	201	303

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_4	0	101	102	201	203	302	303	404

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_4	0	101	201	302	404

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

i

n = 4

L_4	0	101	201	302

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

n = 4

z*

L_4	0	101	201	302

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

An Example of APPROX-SUBSET-SUM

APPROX-SUBSET-SUM(S, t, ε)

1. n = |S|
2. L_0=<0>
3. for i = 1 to n
4. L_i = MERGE-LISTS(L_{i-1},L_{i-1}+x_i)
5. L_i = TRIM(L_i, ε/2n)
6. remove from L_i every element that is greater than t
7. let z* be the largest value in L_n
8. return z*

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

n = 4

z*

L_4	0	101	201	302

An Example of APPROX-SUBSET-SUM

- The algorithm returns z* = 302 as its answer, which is well within ε = 40% of the optimal answer 307 = 104 + 102 + 101.
- In fact, it is within 2%.

t = 308, ε=0.4 , δ=ε/2n=0.05

S	104	102	201	101

z*

L_4	0	101	201	302

APPROX-SUBSET-SUM

- Theorem 35.8
  - APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.
- Proof
  - The operations of trimming L_i in line 5 and removing from L_i every element that is greater than t maintain the property that every element of L_i is also a member of P_i.
  - Therefore, the value z^*returned in line 8 is indeed the sum of some subset of S.

APPROX-SUBSET-SUM

- Proof - continued
  - Let $y^* \in P_n$ denote an optimal solution to the subset-sum problem.
  - Then, from line 6, we know that $z^* \leq y^*$.
  - We need to show that

$$\max\left(\frac{y^*}{z^*}, \frac{z^*}{y^*}\right) = \frac{y^*}{z^*} \leq 1 + \epsilon.$$

  - We must also show that the running time of this algorithm is polynomial in both $1/\epsilon$ and the size of the input (# of bits log t needed to represent t + # of bits needed to represent the set S).

Proof of $\frac{y^*}{z^*} \leq 1 + \epsilon$

- Since $z^* \leq y^*$, we will show that $\frac{y^*}{z^*} \leq 1 + \epsilon$
  - z^* : approximate solution
  - y^* : optimal solution
- $\forall y \in P_i$, where $y \leq t$, $\exists z \in L_i$ s.t. $\frac{y}{\left(1+\frac{\epsilon}{2n}\right)^i} \leq z \leq y$ (Exercise 35.5-2)
  - P_i : the i-th set of all possible summation values
  - L_i : the i-th trimmed set
- Thus, for $y^* \in P_n$, we have $z \in L_n$ s.t.

$$\frac{y^*}{\left(1+\frac{\epsilon}{2n}\right)^n} \leq z \leq y^*.$$

- Since there exists an element $z \in L_n$ fulfilling the above inequality, the inequality must hold for z^*, which is the largest value in L_n. That is,

$$\frac{y^*}{z^*} \leq \left(1 + \frac{\epsilon}{2n}\right)^n.$$

Proof of $\frac{y^*}{z^*} \leq 1 + \epsilon$

- Since $\frac{y^*}{z^*} \leq \left(1 + \frac{\epsilon}{2n}\right)^n$, to show that $\frac{y^*}{z^*} \leq 1 + \epsilon$, we prove that $\left(1 + \frac{\epsilon}{2n}\right)^n \leq 1 + \epsilon$.
- The function $(1+ \epsilon/2n)^n$ increases with n as it approaches its limit of $e^{\epsilon/2}$ since
  - $\left(1 + \frac{\epsilon}{2n}\right)^n$ monotonically increases ($\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0$)
  - $\lim_{n\to\infty} \left(1 + \frac{\epsilon}{2n}\right)^n = e^{\epsilon/2}$ (By equation (3.14)).
- Now, we have $\left(1 + \frac{\epsilon}{2n}\right)^n \leq e^{\epsilon/2}$

$$\leq 1 + \frac{\epsilon}{2} + \left(\frac{\epsilon}{2}\right)^2 \qquad \text{(by inequality 3.13)}$$

$$\leq 1 + \epsilon.$$

- Thus, y*/z* $\leq 1 + \epsilon$.

$$1 + x \leq e^x \leq 1 + x + x^2$$

Proof of Time Complexity

- We will show that a bound of $|L_i|$ is polynomial in the size of input and $\frac{1}{\epsilon}$.
- After trimming, successive element z and z' must have the relationship $\frac{z'}{z} > 1 + \frac{\epsilon}{2n}$.
- Therefore, L_i contains 0, possibly the value 1, and up to additional $\lfloor \log_{1+\epsilon/2n} t \rfloor$ values because
  - $|L_i| < |\{0,1,1 + \frac{\epsilon}{2n}, \left(1 + \frac{\epsilon}{2n}\right)^2, \dots, \left(1 + \frac{\epsilon}{2n}\right)^k\}|$ where k is the smallest integer s.t. $t < \left(1 + \frac{\epsilon}{2n}\right)^{k+1}$.
  - Thus, $\log_{1+\epsilon/2n} t < k + 1$.
  - That is, $\lfloor \log_{1+\epsilon/2n} t \rfloor \leq k$.

Proof of Time Complexity

- From the previous observation, $|L_i|$ is at most

$$\left(\log_{1+\frac{\epsilon}{2n}} t\right) + 2 = \frac{\ln(t)}{\ln(1+\frac{\epsilon}{2n})} + 2$$

$$\leq \frac{\ln(t)}{\frac{\frac{\epsilon}{2n}}{1+\frac{\epsilon}{2n}}} + 2 \qquad \text{(By inequality 3.17)}$$

$$\frac{x}{1+x} \leq \ln(1+x) \leq x$$

$$= \frac{2n\left(1+\frac{\epsilon}{2n}\right)\ln(t)}{\epsilon} + 2$$

$$= \frac{(2n+\epsilon)\ln(t)}{\epsilon} + 2$$

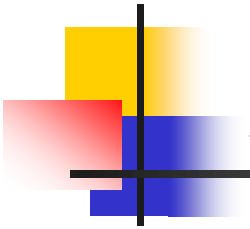
$$< \frac{1}{\epsilon}\left(3n * \ln(t)\right) + 2.$$

$1 - \frac{1}{t} \leq \ln t$ for $t > 0$
$\Rightarrow t \leftarrow 1 + x$ then
$\therefore \frac{x}{1+x} \leq \ln(1+x)$

$0 < \epsilon < 1 < n$

Proof of Time Complexity

- With $|L_i| = O(\frac{1}{\varepsilon} \cdot n \cdot \ln t)$, the bound is polynomial in the size of the input $\ln t$, $\frac{1}{\varepsilon}$ and n.
- Since the running time of APPROX-SUBSET-SUM is polynomial in $|L_i|$, we can conclude that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme.



Any Question?