Coping with NP-Completeness

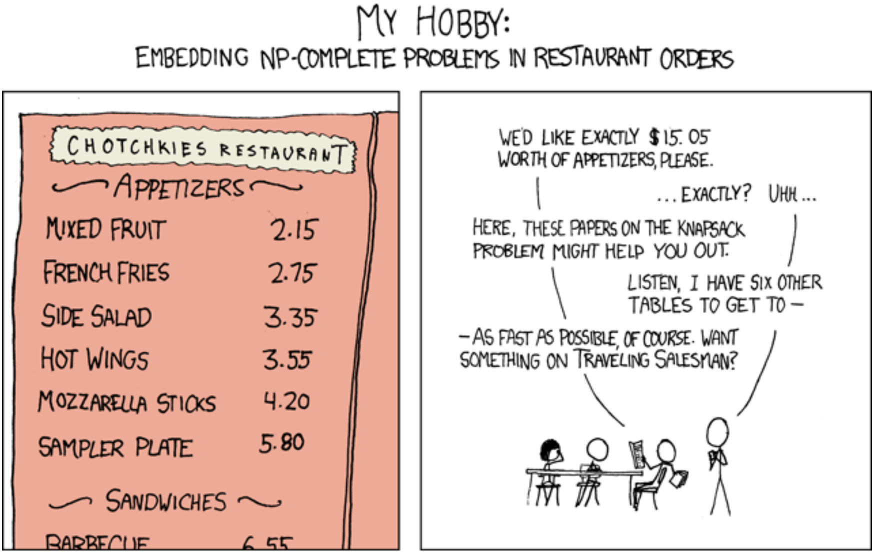
T. M. Murali

May 2, 7, 2013

How Do We Tackle an $\mathcal{NP}$-Complete Problem?

- These problems come up in real life.

How Do We Tackle an $\mathcal{NP}$-Complete Problem?



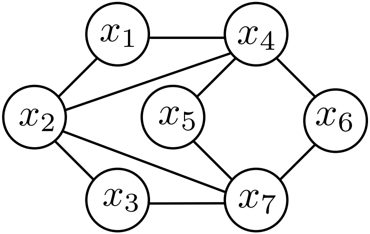
How Do We Tackle an $\mathcal{NP}$-Complete Problem?

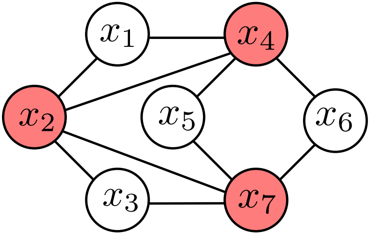
- These problems come up in real life.
- $\mathcal{NP}$-Complete means that a problem is hard to solve in the *worst case*. Can we come up with better solutions at least in *some* cases?

How Do We Tackle an $\mathcal{NP}$-Complete Problem?

- These problems come up in real life.
- $\mathcal{NP}$-Complete means that a problem is hard to solve in the *worst case*. Can we come up with better solutions at least in *some* cases?
  - Develop algorithms that are exponential in one parameter in the problem.
  - Consider special cases of the input, e.g., graphs that "look like" trees.
  - Develop algorithms that can provably compute a solution close to the optimal.

Vertex Cover Problem





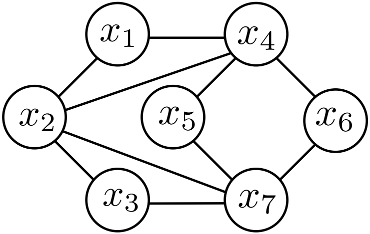
VERTEX COVER

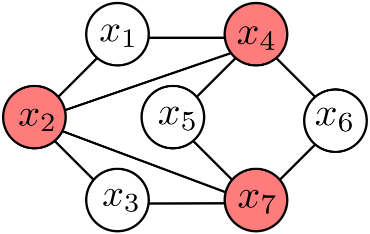
INSTANCE: Undirected graph G and an integer k

QUESTION: Does G contain a vertex cover of size at most k?

- The problem has two parameters: k and n, the number of nodes in G.
- What is the running time of a brute-force algorithm?

Vertex Cover Problem





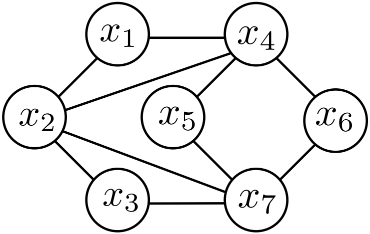
VERTEX COVER

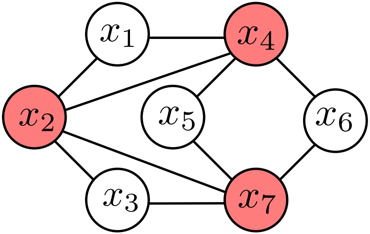
INSTANCE: Undirected graph G and an integer k

QUESTION: Does G contain a vertex cover of size at most k?

- The problem has two parameters: k and n, the number of nodes in G.
- What is the running time of a brute-force algorithm? $O(kn\binom{n}{k}) = O(kn^{k+1})$.

Vertex Cover Problem





VERTEX COVER

INSTANCE: Undirected graph G and an integer k

QUESTION: Does G contain a vertex cover of size at most k?

- The problem has two parameters: k and n, the number of nodes in G.
- What is the running time of a brute-force algorithm? $O(kn\binom{n}{k}) = O(kn^{k+1})$.
- Can we devise an algorithm whose running time is exponential in k but polynomial in n, e.g., $O(2^k n)$?

Designing the Vertex Cover Algorithm

- Intution: if a graph has a small vertex cover, it cannot have too many edges.

Designing the Vertex Cover Algorithm

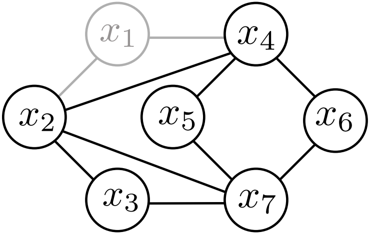
- Intution: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If G has n nodes and G has a vertex cover of size at most k, then G has at most kn edges.

Designing the Vertex Cover Algorithm

- Intution: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If G has n nodes and G has a vertex cover of size at most k, then G has at most kn edges.
- Easy part of algorithm: Return no if G has more than kn edges.

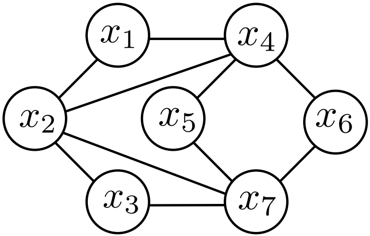
Designing the Vertex Cover Algorithm

- Intution: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If G has n nodes and G has a vertex cover of size at most k, then G has at most kn edges.
- Easy part of algorithm: Return no if G has more than kn edges.
- $G - \{u\}$ is the graph G without node u and the edges incident on u.



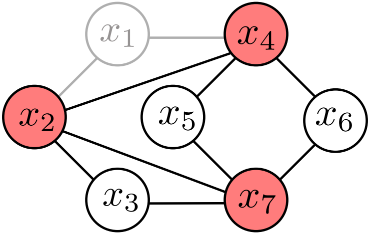
Designing the Vertex Cover Algorithm

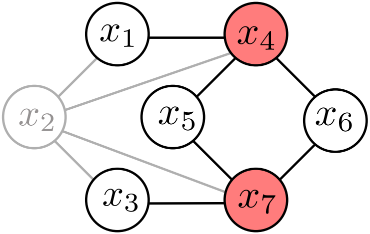
- Intution: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If G has n nodes and G has a vertex cover of size at most k, then G has at most kn edges.
- Easy part of algorithm: Return no if G has more than kn edges.
- $G - \{u\}$ is the graph G without node u and the edges incident on u.
- Consider an edge (u, v). Either u or v must be in the vertex cover.



Designing the Vertex Cover Algorithm

- Intution: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If G has n nodes and G has a vertex cover of size at most k, then G has at most kn edges.
- Easy part of algorithm: Return no if G has more than kn edges.
- $G - \{u\}$ is the graph G without node u and the edges incident on u.
- Consider an edge (u, v). Either u or v must be in the vertex cover.
- Claim: G has a vertex cover of size at most k iff for any edge (u, v) either $G - \{u\}$ or $G - \{v\}$ has a vertex cover of size at most $k - 1$.





Vertex Cover Algorithm

To search for a k-node vertex cover in G:
  If G contains no edges, then the empty set is a vertex cover
  If G contains $> k\ |V|$ edges, then it has no k-node vertex cover
  Else let $e = (u, v)$ be an edge of G
    Recursively check if either of $G - \{u\}$ or $G - \{v\}$
        has a vertex cover of size $k - 1$
    If neither of them does, then G has no k-node vertex cover
    Else, one of them (say, $G - \{u\}$) has a $(k-1)$-node vertex cover T
      In this case, $T \cup \{u\}$ is a k-node vertex cover of G
    Endif
  Endif

Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters

Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters n and k.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of VERTEX COVER with parameters n and k.

Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters n and k.
- Let $T(n,k)$ denote the worst-case running time of the algorithm on an instance of VERTEX COVER with parameters n and k.
- $T(n,1) \leq cn$.

Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters n and k.
- Let $T(n,k)$ denote the worst-case running time of the algorithm on an instance of Vertex Cover with parameters n and k.
- $T(n,1) \leq cn$.
- $T(n,k) \leq 2T(n,k-1) + ckn$.
  - We need $O(kn)$ time to count the number of edges.

Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters n and k.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of Vertex Cover with parameters n and k.
- $T(n, 1) \leq cn$.
- $T(n, k) \leq 2T(n, k-1) + ckn$.
  - We need $O(kn)$ time to count the number of edges.
- Claim: $T(n, k) = O(2^k kn)$.

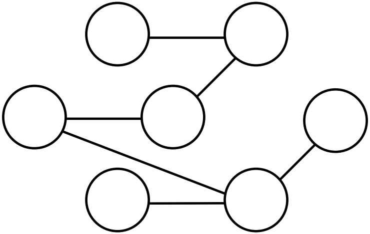
Solving $\mathcal{NP}$-Hard Problems on Trees

- "$\mathcal{NP}$-Hard": at least as hard as $\mathcal{NP}$-Complete. We will use $\mathcal{NP}$-Hard to refer to optimisation versions of decision problems.

Solving $\mathcal{NP}$-Hard Problems on Trees

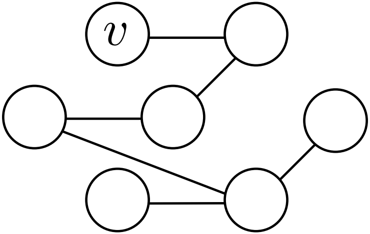
- "$\mathcal{NP}$-Hard": at least as hard as $\mathcal{NP}$-Complete. We will use $\mathcal{NP}$-Hard to refer to optimisation versions of decision problems.
- Many $\mathcal{NP}$-Hard problems can be solved efficiently on trees.
- Intuition: subtree rooted at any node v of the tree "interacts" with the rest of tree only through v. Therefore, depending on whether we include v in the solution or not, we can decouple solving the problem in v's subtree from the rest of the tree.

Designing Greedy Algorithm for Independent Set



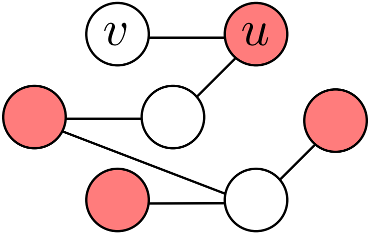
- Optimisation problem: Find the largest independent set in a tree.

Designing Greedy Algorithm for Independent Set



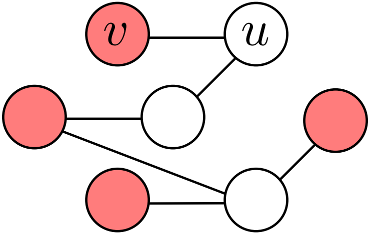
- Optimisation problem: Find the largest independent set in a tree.
- Claim: Every tree $T(V, E)$ has a *leaf*, a node with degree 1.
- Claim: If a tree T has a leaf v, then there exists a maximum-size independent set in T that contains v.

Designing Greedy Algorithm for Independent Set



- Optimisation problem: Find the largest independent set in a tree.
- Claim: Every tree $T(V, E)$ has a *leaf*, a node with degree 1.
- Claim: If a tree T has a leaf v, then there exists a maximum-size independent set in T that contains v. Prove by exchange argument.
  - Let S be a maximum-size independent set that does not contain v.
  - Let v be connected to u.
  - u must be in S; otherwise, we can add v to S, which means S is not maximum size.
  - Since u is in S, we can swap u and v.

Designing Greedy Algorithm for Independent Set



- Optimisation problem: Find the largest independent set in a tree.
- Claim: Every tree $T(V, E)$ has a *leaf*, a node with degree 1.
- Claim: If a tree T has a leaf v, then there exists a maximum-size independent set in T that contains v. Prove by exchange argument.
  - Let S be a maximum-size independent set that does not contain v.
  - Let v be connected to u.
  - u must be in S; otherwise, we can add v to S, which means S is not maximum size.
  - Since u is in S, we can swap u and v.
- Claim: If a tree T has a a leaf v, then a maximum-size independent set in T is v and a maximum-size independent set in $T - \{v\}$.

Greedy Algorithm for Independent Set

- A *forest* is a graph where every connected component is a tree.

```
To find a maximum-size independent set in a forest F:
  Let S be the independent set to be constructed (initially empty)
  While F has at least one edge
     Let e = (u, v) be an edge of F such that v is a leaf
     Add v to S
     Delete from F nodes u and v, and all edges incident to them
  Endwhile
  Return S
```

Greedy Algorithm for Independent Set

- A *forest* is a graph where every connected component is a tree.
- Running time of the algorithm is $O(n)$.

```
To find a maximum-size independent set in a forest F:
  Let S be the independent set to be constructed (initially empty)
  While F has at least one edge
     Let e = (u, v) be an edge of F such that v is a leaf
     Add v to S
     Delete from F nodes u and v, and all edges incident to them
  Endwhile
  Return S
```

Greedy Algorithm for Independent Set

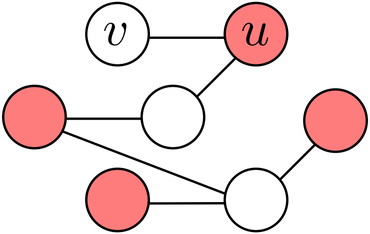
- A *forest* is a graph where every connected component is a tree.
- Running time of the algorithm is $O(n)$.
- The algorithm works correctly on any graph for which we can repeatedly find a leaf.

```
To find a maximum-size independent set in a forest F:
  Let S be the independent set to be constructed (initially empty)
  While F has at least one edge
     Let e = (u, v) be an edge of F such that v is a leaf
     Add v to S
     Delete from F nodes u and v, and all edges incident to them
  Endwhile
  Return S
```

Maximum Weight Independent Set

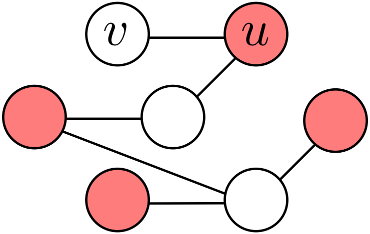
- Consider the INDEPENDENT SET problem but with a weight w_v on every node v.
- Goal is to find an independent set S such that $\sum_{v \in S} w_v$ is as large as possible.

Maximum Weight Independent Set



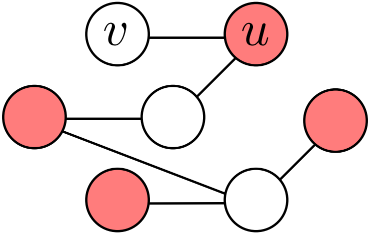
- Consider the INDEPENDENT SET problem but with a weight w_v on every node v.
- Goal is to find an independent set S such that $\sum_{v \in S} w_v$ is as large as possible.
- Can we extend the greedy algorithm?

Maximum Weight Independent Set



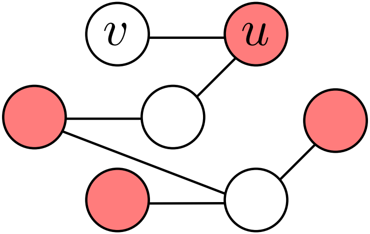
- Consider the INDEPENDENT SET problem but with a weight w_v on every node v.
- Goal is to find an independent set S such that $\sum_{v \in S} w_v$ is as large as possible.
- Can we extend the greedy algorithm? Exchange argument fails: if u is a parent of a leaf v, w_u may be larger than w_v.

Maximum Weight Independent Set



- Consider the Independent Set problem but with a weight w_v on every node v.
- Goal is to find an independent set S such that $\sum_{v \in S} w_v$ is as large as possible.
- Can we extend the greedy algorithm? Exchange argument fails: if u is a parent of a leaf v, w_u may be larger than w_v.
- But there are still only two possibilities: either include u in the independent set or include *all* neighbours of u that are leaves.

Maximum Weight Independent Set



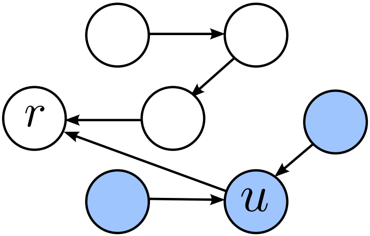
- Consider the INDEPENDENT SET problem but with a weight w_v on every node v.
- Goal is to find an independent set S such that $\sum_{v \in S} w_v$ is as large as possible.
- Can we extend the greedy algorithm? Exchange argument fails: if u is a parent of a leaf v, w_u may be larger than w_v.
- But there are still only two possibilities: either include u in the independent set or include *all* neighbours of u that are leaves.
- Suggests dynamic programming algorithm.

Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.
- What are the sub-problems?

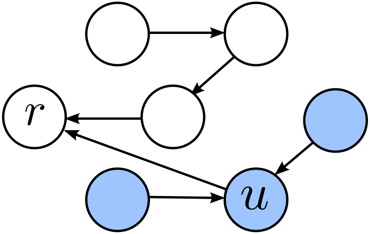
Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.
- What are the sub-problems?
  - Pick a node r and *root* tree at r: orient edges towards r.
  - *parent* $p(u)$ of a node u is the node adjacent to u along the path to r.
  - Sub-problems are T_u: subtree induced by u and all its descendants.

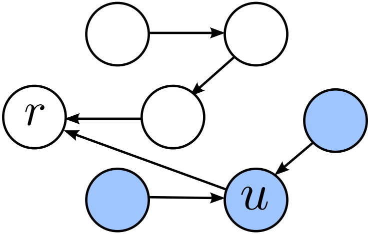


Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.
- What are the sub-problems?
  - Pick a node r and *root* tree at r: orient edges towards r.
  - *parent* $p(u)$ of a node u is the node adjacent to u along the path to r.
  - Sub-problems are T_u: subtree induced by u and all its descendants.
- Ordering the sub-problems: start at leaves and work our way up to the root.

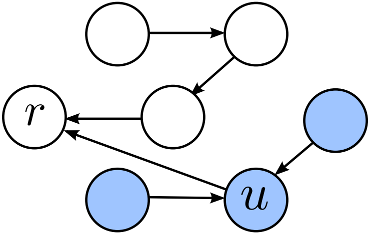


Recursion for Dynamic Programming Algorithm



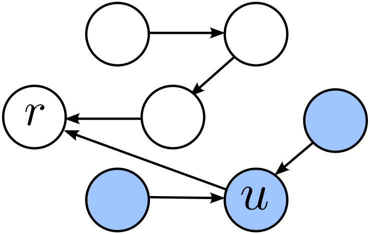
- Either we include u in an optimal solution or exclude u.
  - $OPT_{in}(u)$: maximum weight of an independent set in T_u that includes u.
  - $OPT_{out}(u)$: maximum weight of an independent set in T_u that excludes u.

Recursion for Dynamic Programming Algorithm



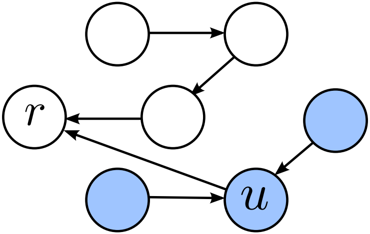
- Either we include u in an optimal solution or exclude u.
  - $OPT_{in}(u)$: maximum weight of an independent set in T_u that includes u.
  - $OPT_{out}(u)$: maximum weight of an independent set in T_u that excludes u.
- Base cases:

Recursion for Dynamic Programming Algorithm



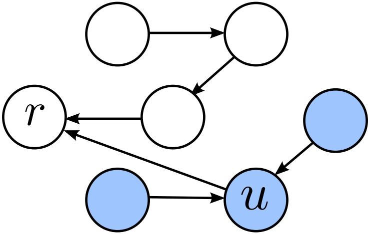
- Either we include u in an optimal solution or exclude u.
  - $OPT_{in}(u)$: maximum weight of an independent set in T_u that includes u.
  - $OPT_{out}(u)$: maximum weight of an independent set in T_u that excludes u.
- Base cases: For a leaf u, $\mathrm{OPT}_{\mathrm{in}}(u) = w_u$ and $\mathrm{OPT}_{\mathrm{out}}(u) = 0$.
- Recurrence: Include u or exclude u.

Recursion for Dynamic Programming Algorithm



- Either we include u in an optimal solution or exclude u.
  - $OPT_{in}(u)$: maximum weight of an independent set in T_u that includes u.
  - $OPT_{out}(u)$: maximum weight of an independent set in T_u that excludes u.
- Base cases: For a leaf u, $\text{OPT}_{\text{in}}(u) = w_u$ and $\text{OPT}_{\text{out}}(u) = 0$.
- Recurrence: Include u or exclude u.
  1. If we include u, all children must be excluded.
     $\text{OPT}_{\text{in}}(u) = w_u + \sum_{v \in \text{children}(u)} \text{OPT}_{\text{out}}(v)$

Recursion for Dynamic Programming Algorithm



- Either we include u in an optimal solution or exclude u.
  - $OPT_{in}(u)$: maximum weight of an independent set in T_u that includes u.
  - $OPT_{out}(u)$: maximum weight of an independent set in T_u that excludes u.
- Base cases: For a leaf u, $\mathrm{OPT}_{\mathrm{in}}(u) = w_u$ and $\mathrm{OPT}_{\mathrm{out}}(u) = 0$.
- Recurrence: Include u or exclude u.
  1. If we include u, all children must be excluded.
     $\mathrm{OPT}_{\mathrm{in}}(u) = w_u + \sum_{v \in \mathrm{children}(u)} \mathrm{OPT}_{\mathrm{out}}(v)$
  2. If we exclude u, a child may or may not be excluded.
     $\mathrm{OPT}_{\mathrm{out}}(u) = \sum_{v \in \mathrm{children}(u)} \max\left(\mathrm{OPT}_{\mathrm{in}}(v), \mathrm{OPT}_{\mathrm{in}}(v)\right)$

Dynamic Programming Algorithm

To find a maximum-weight independent set of a tree T:

Root the tree at a node r

For all nodes u of T in post-order

If u is a leaf then set the values:

$$M_{out}[u] = 0$$

$$M_{in}[u] = w_u$$

Else set the values:

$$M_{out}[u] = \sum_{v \in children(u)} \max(M_{out}[v], \ M_{in}[v])$$

$$M_{in}[u] = \ w_u \ + \sum_{v \in children(u)} M_{out}[u].$$

Endif

Endfor

Return $\max(M_{out}[r], M_{in}[r])$

Dynamic Programming Algorithm

To find a maximum-weight independent set of a tree T:

Root the tree at a node r

For all nodes u of T in post-order

If u is a leaf then set the values:

$M_{out}[u] = 0$

$M_{in}[u] = w_u$

Else set the values:

$$M_{out}[u] = \sum_{v \in children(u)} \max(M_{out}[v],\ M_{in}[v])$$

$$M_{in}[u] = w_u + \sum_{v \in children(u)} M_{out}[u].$$

Endif

Endfor

Return $\max(M_{out}[r], M_{in}[r])$

- Running time of the algorithm is $O(n)$.

Aren't Trees Too Restrictive?

- Trees are only a very specific sub-class of graphs. What use are algorithms for $\mathcal{NP}$-Hard problems that work well on trees?

Aren't Trees Too Restrictive?

- Trees are only a very specific sub-class of graphs. What use are algorithms for $\mathcal{NP}$-Hard problems that work well on trees?
- These ideas can be generalised to graphs that "look like" trees: graphs with bounded treewidth.

Example of Tree Decomposition

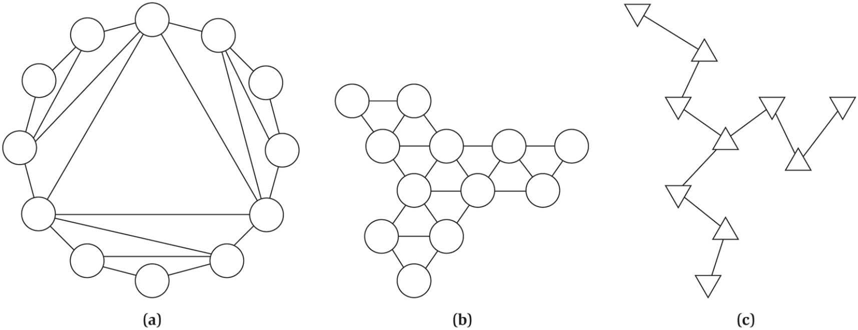


Figure 10.5 Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles "fit together."

Example of Tree Decomposition

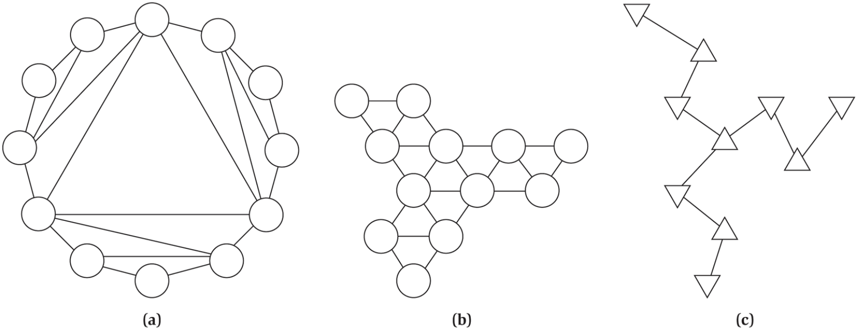


Figure 10.5 Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles "fit together."

- Definition of "tree-like" should capture graphs that we can decompose into disconnected pieces by removing a small number of nodes.
- Definition should make precise the notion of "tree-like" structures in the figure.

Tree Decompositions

A *Tree decomposition* of a graph $G(V, E)$ consists of

1. a tree T (whose nodes are different from V)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$

Tree Decompositions

A *Tree decomposition* of a graph $G(V, E)$ consists of

1. a tree T (whose nodes are different from V)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$

that satisfy three properties:

Tree Decompositions

A *Tree decomposition* of a graph $G(V, E)$ consists of

1. a tree T (whose nodes are different from V)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$

that satisfy three properties:

(Node coverage): Every node of G belongs to at least one piece V_t

Tree Decompositions

A *Tree decomposition* of a graph $G(V, E)$ consists of

1. a tree T (whose nodes are different from V)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$

that satisfy three properties:

(Node coverage): Every node of G belongs to at least one piece V_t
(Edge coverage): For every edge (u, v) in G, there is at least one piece V_t that contains both u and v, and

Tree Decompositions

A *Tree decomposition* of a graph $G(V, E)$ consists of

1. a tree T (whose nodes are different from V)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$

that satisfy three properties:

(Node coverage): Every node of G belongs to at least one piece V_t
(Edge coverage): For every edge (u, v) in G, there is at least one piece V_t that contains both u and v, and
(Coherence): Let t_1, t_2, and t_3 be three nodes in T such that t_2 lies on the path from t_1 to t_3. Then, if a node v in G belongs to V_{t_1} and V_{t_3}, it also belongs to V_{t_2}.

Properties of Tree Decompositions

- Trees have two nice separation properties:
  1. If we delete an edge from a tree, the tree splits into two connected components.
  2. If we delete a node and all incident edges from a tree, the tree splits into a number of connected components equal to the degree of the node.
- Tree decompositions have analogous properties.

Uses of Tree Decompositions

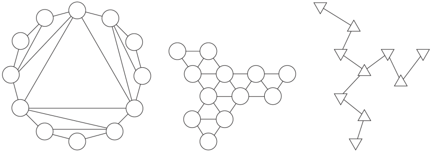


Figure 10.5 Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles "fit together."

- *Width* of a tree decomposition is the size of the largest piece.
- *Treewidth* of a graph is the smallest width of a tree decomposition of the graph.
- If we have a tree decomposition of small width, we can perform dynamic programming over the decomposition.
- Cost of the algorithm is exponential in the width of the decomposition.

Uses of Tree Decompositions

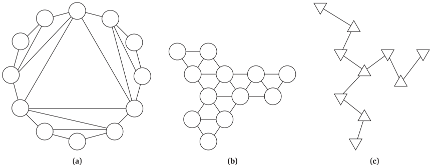


Figure 10.5 Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles "fit together."

- *Width* of a tree decomposition is the size of the largest piece.
- *Treewidth* of a graph is the smallest width of a tree decomposition of the graph.
- If we have a tree decomposition of small width, we can perform dynamic programming over the decomposition.
- Cost of the algorithm is exponential in the width of the decomposition.
- Does a graph a tree decomposition with width at most w?

Uses of Tree Decompositions

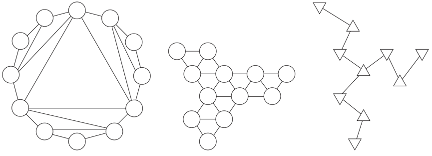


Figure 10.5 Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles "fit together."

- *Width* of a tree decomposition is the size of the largest piece.
- *Treewidth* of a graph is the smallest width of a tree decomposition of the graph.
- If we have a tree decomposition of small width, we can perform dynamic programming over the decomposition.
- Cost of the algorithm is exponential in the width of the decomposition.
- Does a graph a tree decomposition with width at most w? $\mathcal{NP}$-Complete!

Uses of Tree Decompositions

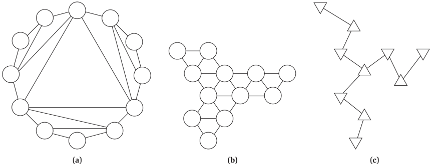


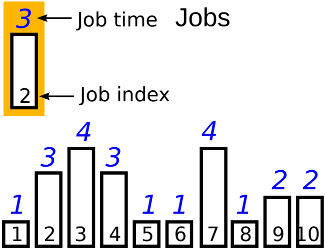
Figure 10.5 Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles "fit together."

- *Width* of a tree decomposition is the size of the largest piece.
- *Treewidth* of a graph is the smallest width of a tree decomposition of the graph.
- If we have a tree decomposition of small width, we can perform dynamic programming over the decomposition.
- Cost of the algorithm is exponential in the width of the decomposition.
- Does a graph a tree decomposition with width at most w? $\mathcal{NP}$-Complete!
- (Chapter 10.5): Given a graph and a parameter w, there is an algorithm that runs in $O(f(w)mn)$ time and either
  1. produces a tree decomposition of width at most $4w$ or
  2. reports correctly that G does not have a tree decomposition with width less than w.

Approximation Algorithms

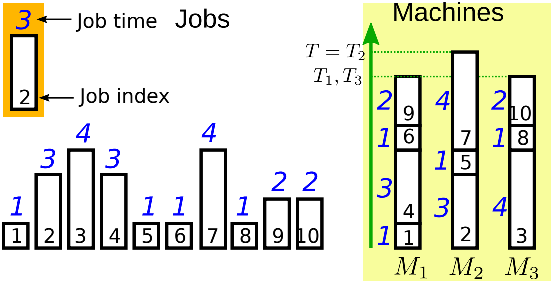
- Methods for optimisation versions of $\mathcal{NP}$-Complete problems.
- Run in polynomial time.
- Solution returned is guaranteed to be within a small factor of the optimal solution

Load Balancing Problem



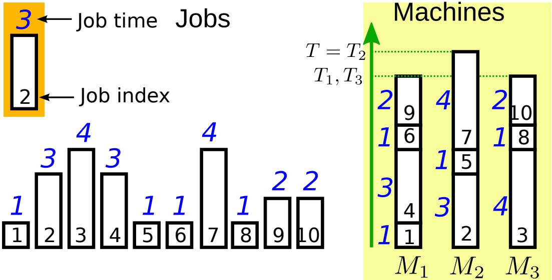
- Given set of m machines $M_1, M_2, \ldots M_m$.
- Given a set of n jobs: job j has processing time t_j.
- Assign each job to one machine so that the total time spent is minimised.

Load Balancing Problem



- Given set of m machines $M_1, M_2, \ldots M_m$.
- Given a set of n jobs: job j has processing time t_j.
- Assign each job to one machine so that the total time spent is minimised.
- Let $A(i)$ be the set of jobs assigned to machine M_i.
- Total time spent on machine i $T_i = \sum_{k \in A(i)} t_k$.
- Minimise *makespan* $T = \max_i T_i$, the largest load on any machine.

Load Balancing Problem



- Given set of m machines $M_1, M_2, \ldots M_m$.
- Given a set of n jobs: job j has processing time t_j.
- Assign each job to one machine so that the total time spent is minimised.
- Let $A(i)$ be the set of jobs assigned to machine M_i.
- Total time spent on machine i $T_i = \sum_{k \in A(i)} t_k$.
- Minimise *makespan* $T = \max_i T_i$, the largest load on any machine.
- Minimising makespan is $\mathcal{NP}$-Complete.

Greedy-Balance Algorithm

- Adopt a greedy approach.
- Process jobs in *any* order.
- Assign next job to the processor that has smallest total load so far.

Greedy-Balance:

Start with no jobs assigned

Set $T_i = 0$ and $A(i) = \emptyset$ for all machines M_i

For $j = 1, \ldots, n$

  Let M_i be a machine that achieves the minimum $\min_k T_k$

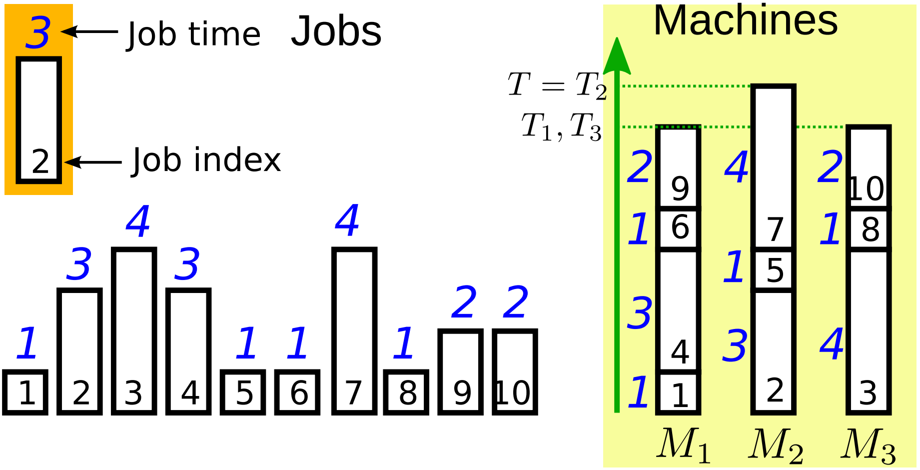
  Assign job j to machine M_i

  Set $A(i) \leftarrow A(i) \cup \{j\}$

  Set $T_i \leftarrow T_i + t_j$

EndFor

Example of Greedy-Balance Algorithm



Lower Bounds on the Optimal Makespan

- We need a lower bound on the optimum makespan T^*.

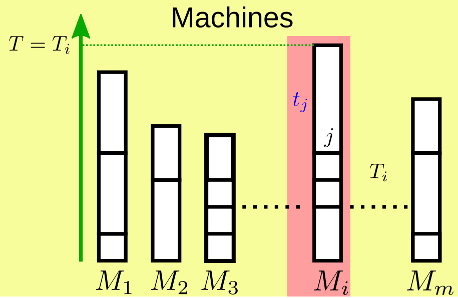
Lower Bounds on the Optimal Makespan

- We need a lower bound on the optimum makespan T^*.
- The two bounds below will suffice:

$$T^* \geq \frac{1}{m} \sum_j t_j$$

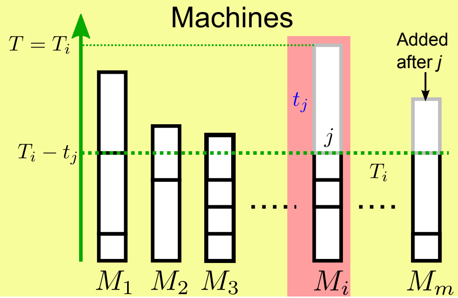
$$T^* \geq \max_j t_j$$

Analysing Greedy-Balance



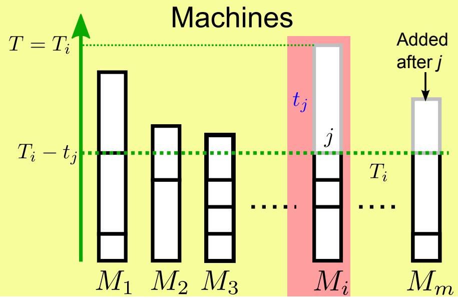
- Claim: Computed makespan $T \leq 2T^*$.

Analysing Greedy-Balance



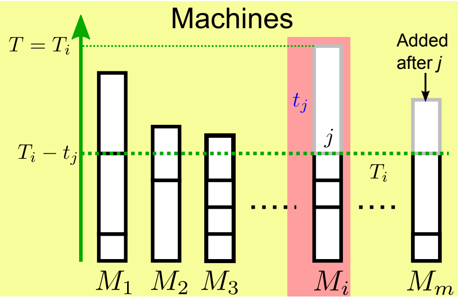
- Claim: Computed makespan $T \leq 2T^*$.
- Let M_i be the machine whose load is T and j be the last job placed on M_i.
- What was the situation just before placing this job?

Analysing Greedy-Balance



- Claim: Computed makespan $T \leq 2T^*$.
- Let M_i be the machine whose load is T and j be the last job placed on M_i.
- What was the situation just before placing this job?
- M_i had the smallest load and its load was $T - t_j$.
- For every machine M_k, load $T_k \geq T - t_j$.

Analysing Greedy-Balance



- Claim: Computed makespan $T \leq 2T^*$.
- Let M_i be the machine whose load is T and j be the last job placed on M_i.
- What was the situation just before placing this job?
- M_i had the smallest load and its load was $T - t_j$.
- For every machine M_k, load $T_k \geq T - t_j$.

$$\sum_k T_k \geq m(T - t_j), \text{ where } k \text{ ranges over machines}$$

$$\sum_j t_j \geq m(T - t_j), \text{ where } j \text{ ranges over jobs}$$

$$T - t_j \leq 1/m \sum_j t_j \leq T^*$$

$$T \leq 2T^*, \text{ since } t_j \leq T^*$$

Improving the Bound

- It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.

Improving the Bound

- It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.
- How can we improve the algorithm?

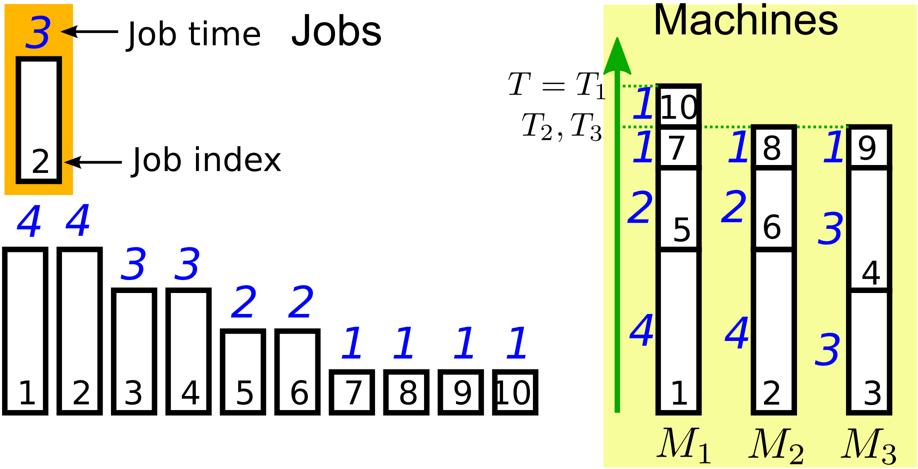
Improving the Bound

- It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.
- How can we improve the algorithm?
- What if we process the jobs in decreasing order of processing time?

Sorted-Balance Algorithm

```
Sorted-Balance:
Start with no jobs assigned
Set T_i = 0 and A(i) = ∅ for all machines M_i
Sort jobs in decreasing order of processing times t_j
Assume that t_1 ≥ t_2 ≥ ... ≥ t_n
For j = 1, ..., n
  Let M_i be the machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
EndFor
```

Example of Sorted-Balance Algorithm



Analyzing Sorted-Balance

- Claim: if there are fewer than m jobs, algorithm is optimal.
- Claim: if there are more than m jobs, then $T^* \geq 2t_{m+1}$.

Analyzing Sorted-Balance

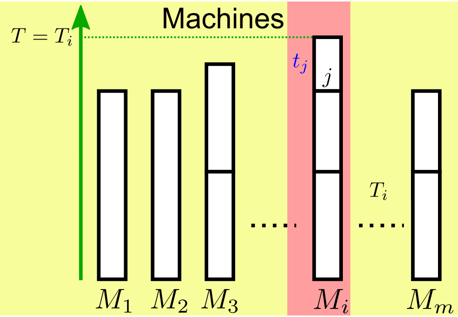
- Claim: if there are fewer than m jobs, algorithm is optimal.
- Claim: if there are more than m jobs, then $T^* \geq 2t_{m+1}$.
  - Consider only the first $m + 1$ jobs in sorted order.
  - Consider *any* assignment of these $m + 1$ jobs to machines.
  - Some machine must be assigned two jobs, each with processing time at least t_{m+1}.
  - This machine will have load at least $2t_{m+1}$.

Analyzing Sorted-Balance

- Claim: if there are fewer than m jobs, algorithm is optimal.
- Claim: if there are more than m jobs, then $T^* \geq 2t_{m+1}$.
  - Consider only the first $m + 1$ jobs in sorted order.
  - Consider *any* assignment of these $m + 1$ jobs to machines.
  - Some machine must be assigned two jobs, each with processing time at least t_{m+1}.
  - This machine will have load at least $2t_{m+1}$.
- Claim: $T \leq 3T^*/2$.

Analyzing Sorted-Balance

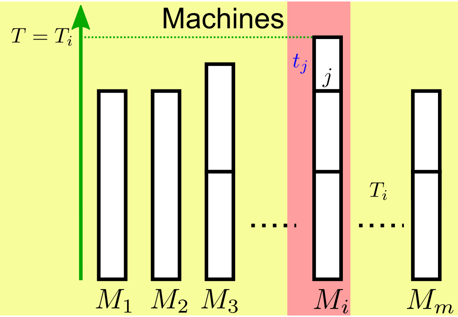
- Claim: if there are fewer than m jobs, algorithm is optimal.
- Claim: if there are more than m jobs, then $T^* \geq 2t_{m+1}$.
  - Consider only the first $m+1$ jobs in sorted order.
  - Consider *any* assignment of these $m+1$ jobs to machines.
  - Some machine must be assigned two jobs, each with processing time at least t_{m+1}.
  - This machine will have load at least $2t_{m+1}$.
- Claim: $T \leq 3T^*/2$.
- Let M_i be the machine whose load is T and j be the last job placed on M_i. (M_i has at least two jobs.)



Analyzing Sorted-Balance

- Claim: if there are fewer than m jobs, algorithm is optimal.
- Claim: if there are more than m jobs, then $T^* \geq 2t_{m+1}$.
  - Consider only the first $m+1$ jobs in sorted order.
  - Consider *any* assignment of these $m+1$ jobs to machines.
  - Some machine must be assigned two jobs, each with processing time at least t_{m+1}.
  - This machine will have load at least $2t_{m+1}$.
- Claim: $T \leq 3T^*/2$.
- Let M_i be the machine whose load is T and j be the last job placed on M_i. (M_i has at least two jobs.)

$$t_j \leq t_{m+1} \leq T^*/2, \text{ since } j \geq m+1$$
$$T - t_j \leq T^*, \text{ Greedy-Balance proof}$$
$$T \leq 3T^*/2$$

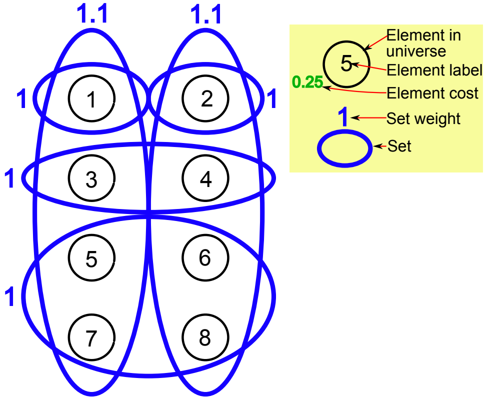


Set Cover

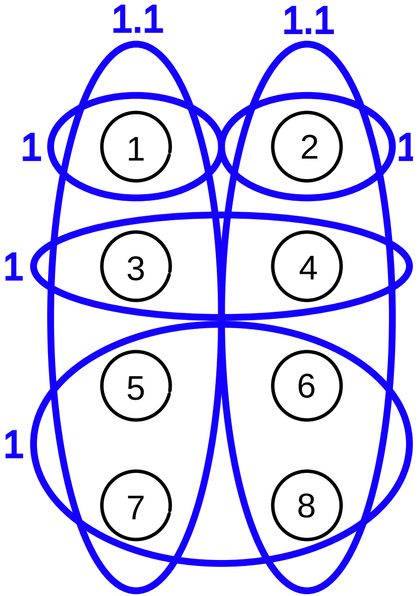
SET COVER

INSTANCE: A set U of n elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, each with an associated weight w.

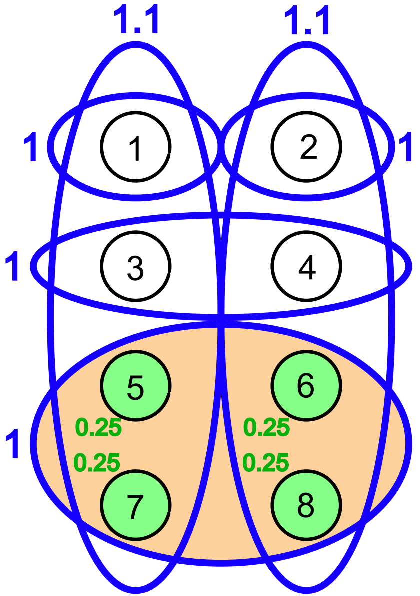
SOLUTION: A collection $\mathcal{C}$ of sets in the collection such that $\bigcup_{S_i \in C} S_i = U$ and $\sum_{S_i \in C} w_i$ is minimised.



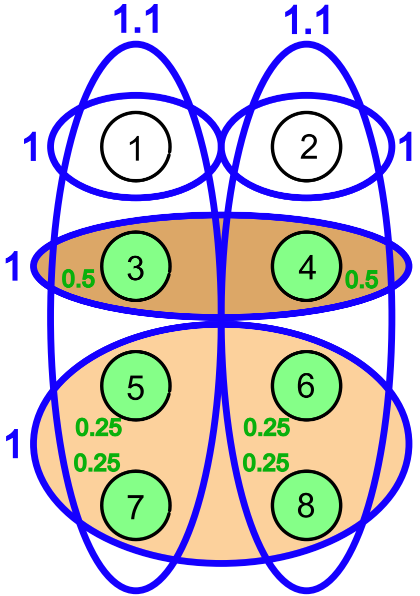
Greedy Approach



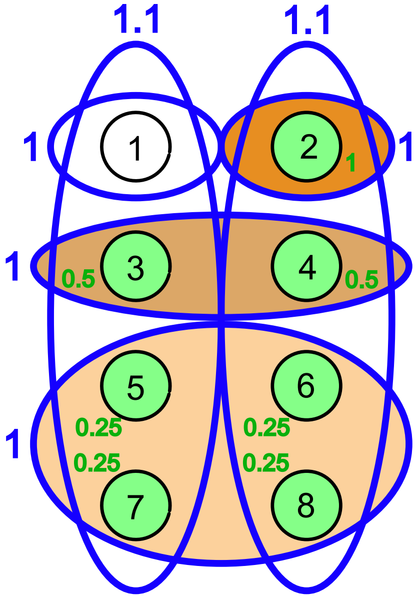
Greedy Approach



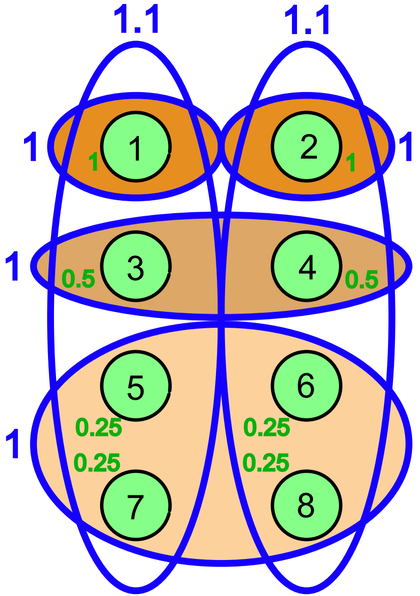
Greedy Approach



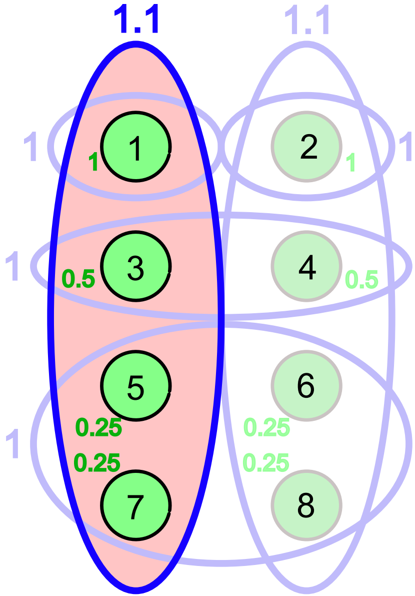
Greedy Approach



Greedy Approach



Greedy Approach



Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?

Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set R of uncovered elements.
- Process set in decreasing order of $w_i/|S_i \cap R|$.

Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set R of uncovered elements.
- Process set in decreasing order of $w_i/|S_i \cap R|$.

Greedy-Set-Cover:

Start with $R = U$ and no sets selected

While $R \neq \emptyset$

  Select set S_i that minimizes $w_i/|S_i \cap R|$

  Delete set S_i from R

EndWhile

Return the selected sets

Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set R of uncovered elements.
- Process set in decreasing order of $w_i/|S_i \cap R|$.

```
Greedy-Set-Cover:
Start with R = U and no sets selected
While R ≠ Ø
  Select set S_i that minimizes w_i/|S_i ∩ R|
  Delete set S_i from R
EndWhile
Return the selected sets
```

- The algorithm computes a set cover whose weight is at most $O(\log n)$ times the optimal weight (Johnson 1974, Lovász 1975, Chvatal 1979).

Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight w^* of the optimum set cover are not easy to obtain.

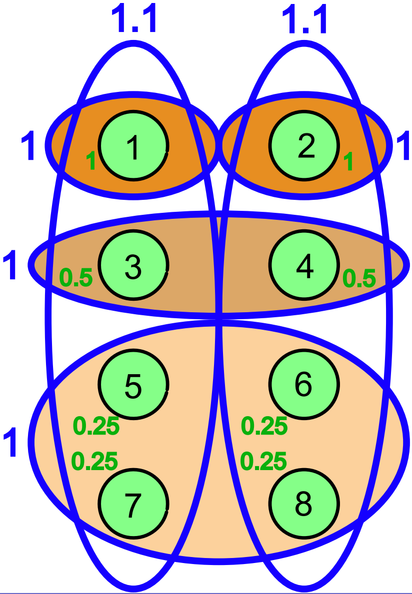
Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight w^* of the optimum set cover are not easy to obtain.
- Bookkeeping: record the per-element *cost* paid when selecting S_i.

Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight w^* of the optimum set cover are not easy to obtain.
- Bookkeeping: record the per-element *cost* paid when selecting S_i.
- In the algorithm, after selecting S_i, add the line

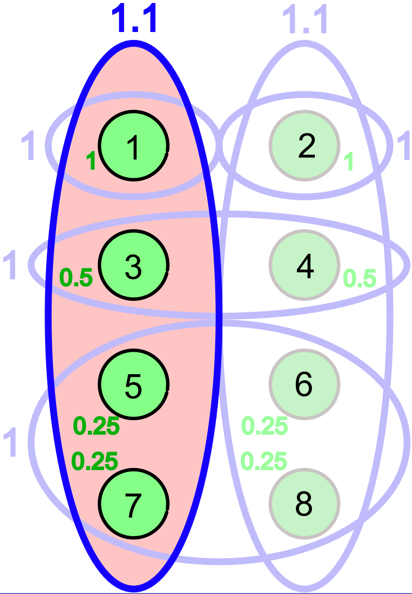
  `Define` $c_s = w_i/|S_i \cap R|$ `for all` $s \in S_i \cap R$.
- As each set S_i is selected, distribute its weight over the costs c_s of the *newly*-covered elements.
- Each element in the universe assigned cost exactly once.



Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight w^* of the optimum set cover are not easy to obtain.
- Bookkeeping: record the per-element *cost* paid when selecting S_i.
- In the algorithm, after selecting S_i, add the line

  `Define` $c_s = w_i/|S_i \cap R|$ `for all` $s \in S_i \cap R$.
- As each set S_i is selected, distribute its weight over the costs c_s of the *newly*-covered elements.
- Each element in the universe assigned cost exactly once.



Starting the Analysis of Greedy-Set-Cover

- Let $\mathcal{C}$ be the set cover computed by Greedy-Set-Cover.
- Claim: $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$.

$$\sum_{S_i \in \mathcal{C}} w_i = \sum_{S_i \in \mathcal{C}} \left(\sum_{s \in S_i \cap R} c_s \right), \text{ by definition of } c_s$$
$$= \sum_{s \in U} c_s, \text{ since each element in the universe contributes exactly once}$$

- In other words, the total weight of the solution computed by Greedy-Set-Cover is the total costs it assigns to the elements in the universe.
- Can "switch" between set-based weight of solution and element-based costs.
- Note: sets have weights whereas Greedy-Set-Cover assigns costs to elements.

Intuition Behind the Proof

- Suppose $\mathcal{C}^*$ is the optimal set cover: $w^* = \sum_{S_j \in \mathcal{C}^*} w_j$.
- Goal is to relate total weight of sets in $\mathcal{C}$ to total weight of sets in $\mathcal{C}^*$.

Intuition Behind the Proof

- Suppose $\mathcal{C}^*$ is the optimal set cover: $w^* = \sum_{S_j \in \mathcal{C}^*} w_j$.
- Goal is to relate total weight of sets in $\mathcal{C}$ to total weight of sets in $\mathcal{C}^*$.
- What is the total cost assigned by GREEDY-SET-COVER to the elements in the sets in the optimal cover $\mathcal{C}^*$?

Intuition Behind the Proof

- Suppose $\mathcal{C}^*$ is the optimal set cover: $w^* = \sum_{S_j \in \mathcal{C}^*} w_j$.
- Goal is to relate total weight of sets in $\mathcal{C}$ to total weight of sets in $\mathcal{C}^*$.
- What is the total cost assigned by GREEDY-SET-COVER to the elements in the sets in the optimal cover $\mathcal{C}^*$?
- Since $\mathcal{C}^*$ is a set cover, $\displaystyle\sum_{S_j \in \mathcal{C}^*} \left(\sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in \mathcal{C}} w_i = w.$

Intuition Behind the Proof

- Suppose $\mathcal{C}^*$ is the optimal set cover: $w^* = \sum_{S_j \in \mathcal{C}^*} w_j$.
- Goal is to relate total weight of sets in $\mathcal{C}$ to total weight of sets in $\mathcal{C}^*$.
- What is the total cost assigned by GREEDY-SET-COVER to the elements in the sets in the optimal cover $\mathcal{C}^*$?
- Since $\mathcal{C}^*$ is a set cover, $\sum_{S_j \in \mathcal{C}^*} \left(\sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in \mathcal{C}} w_i = w$.
- In the sum on the left, S_j is a set in $\mathcal{C}^*$ (need not be a set in $\mathcal{C}$). How large can total cost of elements in such a set be?

Intuition Behind the Proof

- Suppose $\mathcal{C}^*$ is the optimal set cover: $w^* = \sum_{S_j \in \mathcal{C}^*} w_j$.
- Goal is to relate total weight of sets in $\mathcal{C}$ to total weight of sets in $\mathcal{C}^*$.
- What is the total cost assigned by GREEDY-SET-COVER to the elements in the sets in the optimal cover $\mathcal{C}^*$?
- Since $\mathcal{C}^*$ is a set cover, $\displaystyle\sum_{S_j \in \mathcal{C}^*} \left(\sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in \mathcal{C}} w_i = w.$
- In the sum on the left, S_j is a set in $\mathcal{C}^*$ (need not be a set in $\mathcal{C}$). How large can total cost of elements in such a set be?
- For *any* set S_k, suppose we can prove $\sum_{s \in S_k} c_s \leq \alpha w_k$, for some fixed $\alpha > 0$, i.e., total cost assigned by GREEDY-SET-COVER to the elements in S_k cannot be much larger than the weight of s_k.

Intuition Behind the Proof

- Suppose $\mathcal{C}^*$ is the optimal set cover: $w^* = \sum_{S_j \in \mathcal{C}^*} w_j$.
- Goal is to relate total weight of sets in $\mathcal{C}$ to total weight of sets in $\mathcal{C}^*$.
- What is the total cost assigned by GREEDY-SET-COVER to the elements in the sets in the optimal cover $\mathcal{C}^*$?
- Since $\mathcal{C}^*$ is a set cover, $$\sum_{S_j \in \mathcal{C}^*} \left(\sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in \mathcal{C}} w_i = w.$$
- In the sum on the left, S_j is a set in $\mathcal{C}^*$ (need not be a set in $\mathcal{C}$). How large can total cost of elements in such a set be?
- For *any* set S_k, suppose we can prove $\sum_{s \in S_k} c_s \leq \alpha w_k$, for some fixed $\alpha > 0$, i.e., total cost assigned by GREEDY-SET-COVER to the elements in S_k cannot be much larger than the weight of s_k.
- Then $$w \leq \sum_{S_j \in \mathcal{C}^*} \left(\sum_{s \in S_j} c_s \right) \leq \sum_{S_j \in \mathcal{C}^*} \alpha w_j = \alpha w^*.$$

Intuition Behind the Proof

- Suppose $\mathcal{C}^*$ is the optimal set cover: $w^* = \sum_{S_j \in \mathcal{C}^*} w_j$.
- Goal is to relate total weight of sets in $\mathcal{C}$ to total weight of sets in $\mathcal{C}^*$.
- What is the total cost assigned by GREEDY-SET-COVER to the elements in the sets in the optimal cover $\mathcal{C}^*$?
- Since $\mathcal{C}^*$ is a set cover, $\displaystyle\sum_{S_j \in \mathcal{C}^*} \left(\sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in \mathcal{C}} w_i = w.$
- In the sum on the left, S_j is a set in $\mathcal{C}^*$ (need not be a set in $\mathcal{C}$). How large can total cost of elements in such a set be?
- For *any* set S_k, suppose we can prove $\sum_{s \in S_k} c_s \leq \alpha w_k$, for some fixed $\alpha > 0$, i.e., total cost assigned by GREEDY-SET-COVER to the elements in S_k cannot be much larger than the weight of s_k.
- Then $\displaystyle w \leq \sum_{S_j \in \mathcal{C}^*} \left(\sum_{s \in S_j} c_s \right) \leq \sum_{S_j \in \mathcal{C}^*} \alpha w_j = \alpha w^*.$
- For every set S_k in the input, goal is to prove an upper bound on $\dfrac{\sum_{s \in S_k} c_s}{w_k}$.

Upper Bounding Cost-by-Weight Ratio

- Consider *any* set S_k (even one not selected by the algorithm).
- How large can $\frac{\sum_{s \in S_k} c_s}{w_k}$ get?

Upper Bounding Cost-by-Weight Ratio

- Consider *any* set S_k (even one not selected by the algorithm).
- How large can $\frac{\sum_{s \in S_k} c_s}{w_k}$ get?
- The *harmonic function*

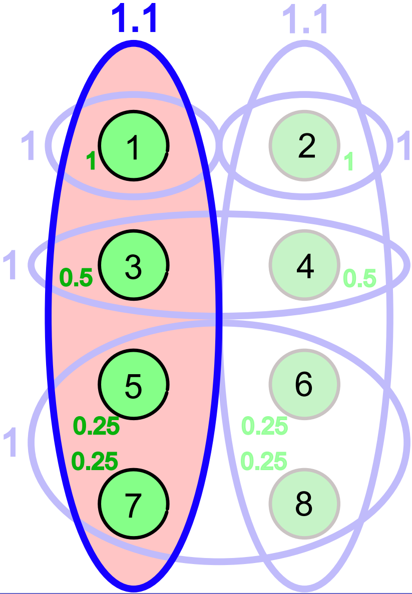
$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n).$$

Upper Bounding Cost-by-Weight Ratio

- Consider *any* set S_k (even one not selected by the algorithm).
- How large can $\frac{\sum_{s \in S_k} c_s}{w_k}$ get?
- The *harmonic function*

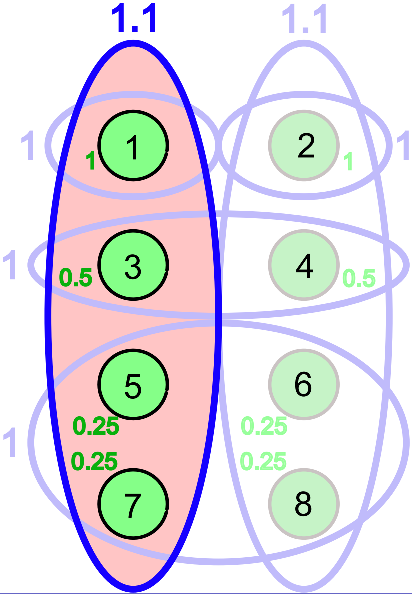
$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n).$$

- Claim: For every set S_k, the sum $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$.



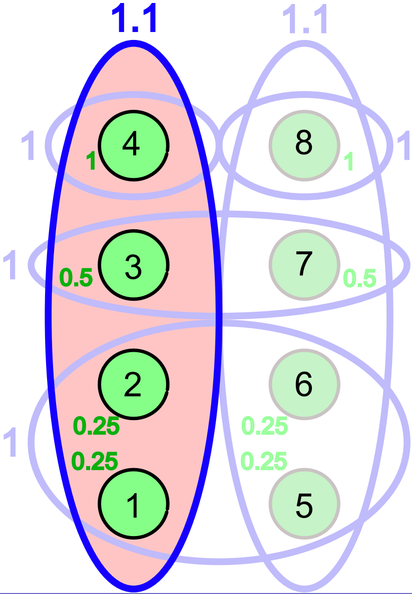
Renumbering Elements in S_k

- Renumber elements in U so that elements in S_k are the first $d = |S_k|$ elements of U, i.e., $S_k = \{s_1, s_2, \ldots, s_d\}$.
- Order elements of S in the order they get covered by the algorithm (i.e., when they get assigned a cost by GREEDY-SET-COVER).



Renumbering Elements in S_k

- Renumber elements in U so that elements in S_k are the first $d = |S_k|$ elements of U, i.e., $S_k = \{s_1, s_2, \ldots, s_d\}$.
- Order elements of S in the order they get covered by the algorithm (i.e., when they get assigned a cost by GREEDY-SET-COVER).

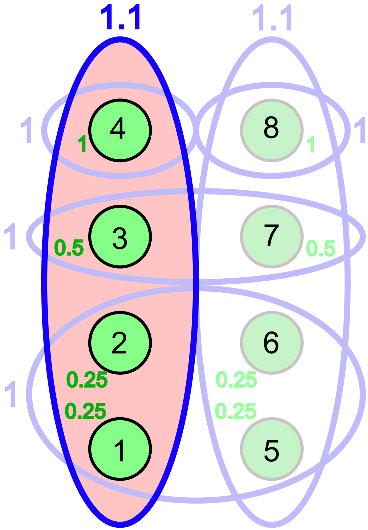


Proving $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?

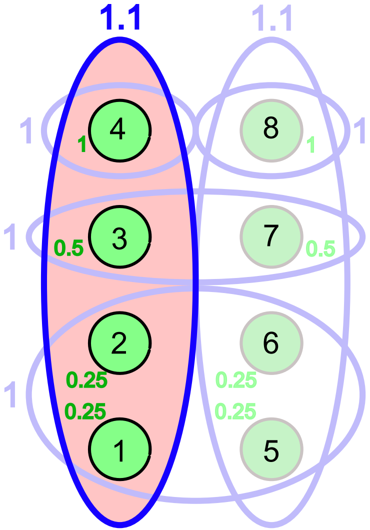
Proving $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?
- At the start of this iteration, R must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of S_k as well.)



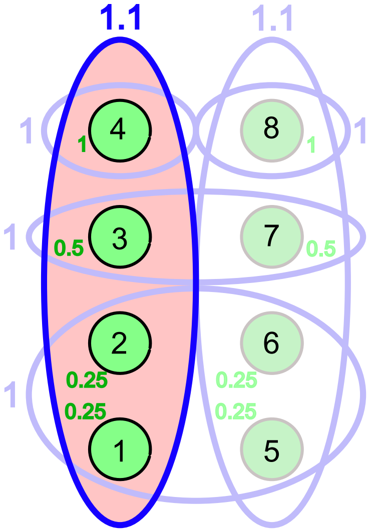
Proving $\sum_{s \in S_k} c_s \leq H(|S_K|) w_k$

- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?
- At the start of this iteration, R must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of S_k as well.)
- Therefore, $\dfrac{w_k}{|S_k \cap R|} \leq \dfrac{w_k}{d - j + 1}$.



Proving $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

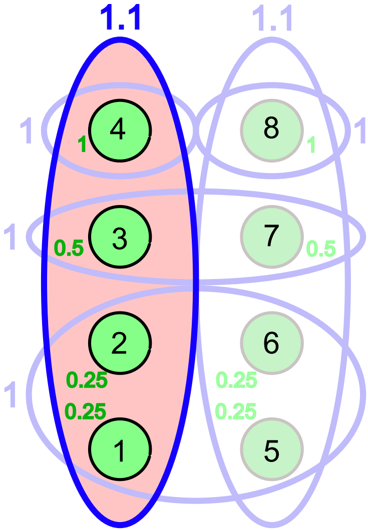
- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?
- At the start of this iteration, R must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of S_k as well.)
- Therefore, $\dfrac{w_k}{|S_k \cap R|} \leq \dfrac{w_k}{d - j + 1}$.
- What cost did the algorithm assign to s_j?
- Suppose the algorithm selected set S_i in this iteration. $c_{s_j} = \dfrac{w_i}{|S_i \cap R|} \leq \dfrac{w_k}{|S_k \cap R|} \leq \dfrac{w_k}{d - j + 1}$.



Proving $\sum_{s \in S_k} c_s \leq H(|S_K|) w_k$

- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?
- At the start of this iteration, R must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of S_k as well.)
- Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.
- What cost did the algorithm assign to s_j?
- Suppose the algorithm selected set S_i in this iteration.
  $c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.
- We are done!

$$\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{d - j + 1} = H(d) w_k.$$



Proving Upper Bound on Cost of Greedy-Set-Cover

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$.
- Let d^* be the size of the largest set in the collection.
- Recall that $\mathcal{C}^*$ is the optimal set cover and $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$.

Proving Upper Bound on Cost of Greedy-Set-Cover

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$.
- Let d^* be the size of the largest set in the collection.
- Recall that $\mathcal{C}^*$ is the optimal set cover and $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$.
- For each set S_j in $\mathcal{C}^*$, we have $w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)}$.
- Combining with $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$, we have

$$w^* = \sum_{S_j \in \mathcal{C}^*} w_j$$

Proving Upper Bound on Cost of Greedy-Set-Cover

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_K|) w_k$.
- Let d^* be the size of the largest set in the collection.
- Recall that $\mathcal{C}^*$ is the optimal set cover and $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$.
- For each set S_j in $\mathcal{C}^*$, we have $w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)}$.
- Combining with $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$, we have

$$w^* = \sum_{S_j \in \mathcal{C}^*} w_j \geq \sum_{S_j \in \mathcal{C}^*} \frac{1}{H(d^*)} \sum_{s \in S_j} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s$$

Proving Upper Bound on Cost of Greedy-Set-Cover

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_K|) w_k$.
- Let d^* be the size of the largest set in the collection.
- Recall that $\mathcal{C}^*$ is the optimal set cover and $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$.
- For each set S_j in $\mathcal{C}^*$, we have $w_j \geq \dfrac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \dfrac{\sum_{s \in S_j} c_s}{H(d^*)}$.
- Combining with $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$, we have

$$w^* = \sum_{S_j \in \mathcal{C}^*} w_j \geq \sum_{S_j \in \mathcal{C}^*} \frac{1}{H(d^*)} \sum_{s \in S_j} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in \mathcal{C}} w_i = w.$$

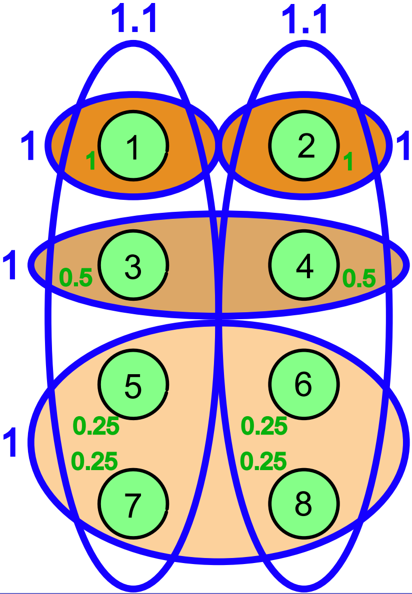
Proving Upper Bound on Cost of Greedy-Set-Cover

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$.
- Let d^* be the size of the largest set in the collection.
- Recall that $\mathcal{C}^*$ is the optimal set cover and $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$.
- For each set S_j in $\mathcal{C}^*$, we have $w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)}$.
- Combining with $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$, we have

$$w^* = \sum_{S_j \in \mathcal{C}^*} w_j \geq \sum_{S_j \in \mathcal{C}^*} \frac{1}{H(d^*)} \sum_{s \in S_j} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in \mathcal{C}} w_i = w.$$

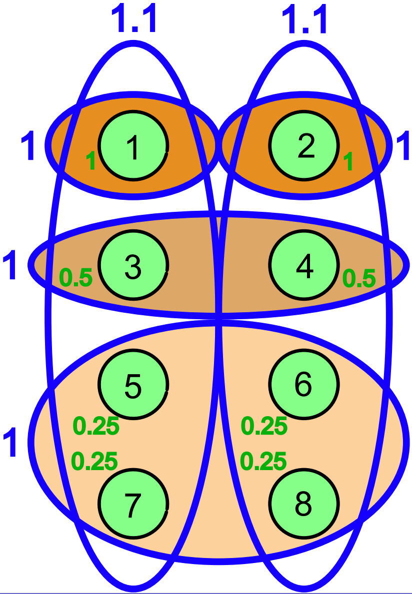
- We have proven that Greedy-Set-Cover computes a set cover whose weight is at most $H(d^*)$ times the optimal weight.

How Badly Can Greedy-Set-Cover Perform?



- Generalise this example to show that algorithm produces a set cover of weight $\Omega(\log n)$ even though optimal weight is $2 + \varepsilon$.
- More complex constructions show greedy algorithm incurs a weight close to $H(n)$ times the optimal weight.

How Badly Can Greedy-Set-Cover Perform?



- Generalise this example to show that algorithm produces a set cover of weight $\Omega(\log n)$ even though optimal weight is $2 + \varepsilon$.
- More complex constructions show greedy algorithm incurs a weight close to $H(n)$ times the optimal weight.
- No polynomial time algorithm can achieve an approximation bound better than $H(n)$ times optimal unless $\mathcal{P} = \mathcal{NP}$ (Lund and Yannakakis, 1994).