Algorithms for Data Processing
Lecture IX: Solving Intractable Problems

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science
http://www.inf.unibz.it/˜artale
`artale@inf.unibz.it`

2019/20 – First Semester
MSc in Computational Data Science — UNIBZ

Some material (text, figures) displayed in these slides is courtesy of:
Alberto Montresor, Werner Nutt, Kevin Wayne, Jon Kleinberg, Eva Tardos.

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard problem. What should I do?

A. Sacrifice one of three desired features.

i. Solve arbitrary instances of the problem.
ii. Solve problem to optimality.
iii. Solve problem in polynomial time.

Coping strategies.

i. Design algorithms for special cases of the problem.
ii. Design approximation algorithms or heuristics.
iii. Design algorithms that may take exponential time.

using greedy, dynamic programming, divide-and-conquer, and network flow algorithms!

Vertex Cover

Definition. Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?

Like many NP-complete problems, Vertex Cover comes with two parameters: n, the nodes in the graph, and k, the size of the vertex cover.

- There are n^k different subsets of V of size k;
- Each takes time $O(kn)$ to check whether it is a vertex cover;
- Thus, in the worst case, the total running time is $O(kn^{k+1})$.
  - if $n = 1,000$ and $k = 10$, even on a PC computing a million of instructions per second we need 10^{24} seconds...which is larger than the age of the UNIVERSE!!!!

Vertex Cover

Definition. Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?

Like many NP-complete problems, Vertex Cover comes with two parameters: n, the nodes in the graph, and k, the size of the vertex cover.

- There are n^k different subsets of V of size k;
- Each takes time $O(kn)$ to check whether it is a vertex cover;
- Thus, in the worst case, the total running time is $O(kn^{k+1})$.
    - if $n = 1,000$ and $k = 10$, even on a PC computing a million of instructions per second we need 10^{24} seconds...which is larger than the age of the UNIVERSE!!!!
- A much better algorithm can be developed, with a running time bound of $O(2^k kn)$.
    - if $n = 1,000$ and $k = 10$, we need few seconds!!!

Vertex Cover/2

Prop.1. If $G = (V, E)$ has n nodes and a vertex cover of size k, then G has at most $k(n-1)$ edges.

Prop.2. Let $e = (u, v)$ be any edge of G. The graph G has a vertex cover of size at most k if and only if at least one of the graphs $G \setminus \{u\}$ and $G \setminus \{v\}$ has a vertex cover of size at most $k-1$.

Vertex Cover – Exact Algorithm

```
VERTEX-COVER(G,k)
if $G$ contains no edges then
    return $T = \emptyset$;                                   /* the empty set is a vertex cover */
if $G$ contains $> k|V|$ edges then
    halt $G$ does not have a $k$-node vertex cover ;          /* this check costs $O$(kn) */
else
    let $e = (u, v)$ be an edge of $G$;
    if $T$=VERTEX-COVER($G \setminus \{u\}, k - 1$) then
        return $T \cup \{u\}$
    if $T$=VERTEX-COVER($G \setminus \{v\}, k - 1$) then
        return $T \cup \{v\}$
```

Vertex Cover – Algorithm Running Time

- The algorithm produces two recursive calls splitting the problem in two sub-problems each with $n-1$ nodes and integer $k-1$.
- In each recursive call we spend $O(kn)$ time.
- We thus have the following recurrence (for some constant c):

$$\begin{aligned} T(n,1) &\leq cn, \\ T(n,k) &\leq 2T(n-1,k-1) + ckn. \end{aligned}$$

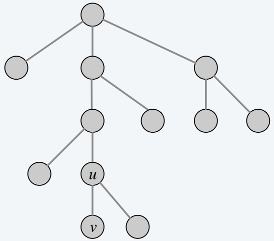
By induction we can show that (see also that the depth of the recursion calls is k):

$$T(n,k) \leq c2^k kn$$

Independent set on trees

Independent set on trees. Given a tree, find a max-cardinality subset of nodes such that no two are adjacent.

Fact. A tree has at least one node that is a leaf (degree = 1).



Key observation. If node v is a leaf, there exists a max-cardinality independent set containing v.

Pf. [exchange argument]

- Consider a max-cardinality independent set S.
- If $v \in S$, we're done.
- Otherwise, let (u, v) denote the lone edge incident to v.
  - if $u \notin S$ and $v \notin S$, then $S \cup \{ v \}$ is independent $\Rightarrow$ S not maximum
  - if $u \in S$ and $v \notin S$, then $S \cup \{ v \} - \{ u \}$ is independent ▪

Independent set on trees: greedy algorithm

Theorem. The greedy algorithm finds a max-cardinality independent set in forests (and hence trees).

Pf. Correctness follows from the previous key observation. ▪

```
INDEPENDENT-SET-IN-A-FOREST($F$)
$S \leftarrow \varnothing$.
WHILE ($F$ has at least 1 edge)
    Let $v$ be a leaf node and let $(u, v)$ be the lone edge incident to $v$.
    $S \leftarrow S \cup \{ v \}$.
    $F \leftarrow F - \{ u, v \}$.  ⟵ delete both $u$ and $v$ (including all incident edges)
RETURN $S \cup \{$ nodes remaining in $F \}$.
```

Remark. Can implement in $O(n)$ time by maintaining nodes of degree 1.

Intractability III: quiz 1

How might the greedy algorithm fail if the graph is not a tree/forest?

A. Might get stuck.

B. Might take exponential time.

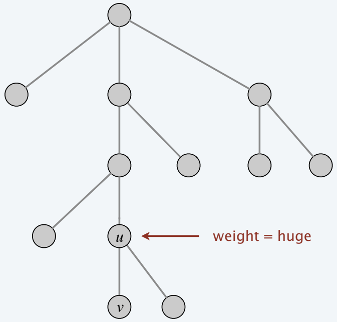
C. Might produce a suboptimal independent set.

D. Any of the above.

Weighted independent set on trees

Weighted independent set on trees. Given a tree and node weights $w_v \geq 0$, find an independent set S that maximizes $\Sigma_{v \in S}\, w_v$.

Greedy algorithm can fail spectacularly.



Weighted Independent Set on Trees – Algorithm

There seems to be no easy way to resolve this problem locally, without considering the rest of the graph, however:

- For the subtree consisting of a node u and its adjacent leaves, we really have only two reasonable solutions to consider:
  1. Including u, or
  2. Including all its adjacent leaves.
- Based on the above idea we build a polynomial algorithm using dynamic programming
  - Record few different solutions, going through a sequence of subproblems, and decide only at the end which of these possibilities is the best overall solution.

Weighted Independent Set on Trees – Dynamic Programming

Start at the leaves and gradually work our way up the tree till we reach the root.

- For a node u, we solve the subproblem associated with the subtree T_u after we have solved the subproblems for all its children.
- The algorithm considers two cases: Either we include the node u in S or we do not.
  1. If we include u, then we cannot include any of its children;
  2. If we do not include u, then we have the freedom to include or omit its children.

Weighted independent set on trees

Weighted independent set on trees. Given a tree and node weights $w_v \geq 0$, find an independent set S that maximizes $\Sigma_{v \in S}\, w_v$.

Dynamic-programming solution. Root tree at some node, say r.

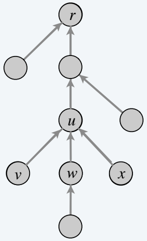
- $OPT_{in}(u)$ = max-weight IS in subtree rooted at u, containing u.
- $OPT_{out}(u)$ = max-weight IS in subtree rooted at u, not containing u.
- Goal: $\max\{\, OPT_{in}(r),\ OPT_{out}(r)\,\}$.

Bellman equation.

overlapping subproblems

$$OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)$$

$$OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max\left\{OPT_{in}(v),\ OPT_{out}(v)\right\}$$



children(u) = { v, w, x }

Weighted independent set on trees: dynamic-programming algorithm

Theorem. The DP algorithm computes max weight of an independent set in a tree in $O(n)$ time.

can also find independent set itself (not just value)

```
WEIGHTED-INDEPENDENT-SET-IN-A-TREE ($T$)

Root the tree $T$ at any node $r$.
$S \leftarrow \varnothing$.
FOREACH (node $u$ of $T$ in postorder/topological order)
    IF ($u$ is a leaf node)
        $M_{in}[u] = w_u$.
        $M_{out}[u] = 0$.
    ELSE
        $M_{in}[u] = w_u + \Sigma_{v \in children(u)} M_{out}[v]$.
        $M_{out}[u] = \Sigma_{v \in children(u)} \max \{ M_{in}[v], M_{out}[v] \}$.
RETURN $\max \{ M_{in}[r], M_{out}[r] \}$.
```

ensures a node is processed after all of its descendants

Exact Algorithms for 3-SAT – Brute Force

Brute force. Given a 3-SAT instance with n variables and m clauses, the brute-force algorithm takes $O(m2^n)$ time.

Proof.

- There are 2^n possible truth assignments to the n variables.
- For each clause, we check whether one of the 3 literals is true, requiring in total $3m$ steps.

Exact Algorithms for 3-SAT – Brute Force

Brute force. Given a 3-SAT instance with n variables and m clauses, the brute-force algorithm takes $O(m2^n)$ time.
Proof.

- There are 2^n possible truth assignments to the n variables.
- For each clause, we check whether one of the 3 literals is true, requiring in total $3m$ steps.

We can build better SAT algorithms!!!

- We show an Algorithm that determines whether there's a satisfying assignment in less time than it would take to enumerate all possible settings of the variables!

Exact Algorithms for 3-SAT

We denote by $\Phi|_{x=TRUE}$ a formula obtained from Φ by applying the following rules:

- All clauses containing x are removed;
- If a clause contains the literal $\neg x$ together with another literal, then $\neg x$ is removed form the clause;
- If a clause consist only of $\neg x$ then $\Phi|_{x=TRUE} \equiv \texttt{FALSE}$.

Exact Algorithms for 3-SAT

We denote by $\Phi|_{x=TRUE}$ a formula obtained from Φ by applying the following rules:

- All clauses containing x are removed;
- If a clause contains the literal $\neg x$ together with another literal, then $\neg x$ is removed form the clause;
- If a clause consist only of $\neg x$ then $\Phi|_{x=TRUE} \equiv \texttt{FALSE}$.

We denote by $\Phi|_{x=FALSE}$ a formula obtained from Φ by applying the following rules:

- All clauses containing $\neg x$ are removed;
- If a clause contains the literal x together with another literal, then x is removed form the clause;
- If a clause consist only of x then $\Phi|_{x=FALSE} \equiv \texttt{FALSE}$.

Exact algorithms for 3-satisfiability

A recursive framework. A 3-SAT formula Φ is either empty or the conjunction of a clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ and a 3-SAT formula Φ' with one fewer clause.

$$\begin{aligned} \Phi &= (\ell_1 \vee \ell_2 \vee \ell_3) \wedge \Phi' \\ &= (\ell_1 \wedge \Phi') \vee (\ell_2 \wedge \Phi') \vee (\ell_3 \wedge \Phi') \\ &= (\Phi' \mid \ell_1 = \mathit{true}) \vee (\Phi' \mid \ell_2 = \mathit{true}) \vee (\Phi' \mid \ell_3 = \mathit{true}) \end{aligned}$$

Notation. $\Phi \mid x = \mathit{true}$ is the simplification of Φ by setting x to true.

Ex.

- $\Phi = (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (w \vee y \vee \neg z) \wedge (\neg x \vee y \vee z)$.
- $\Phi' = (x \vee \neg y \vee z) \wedge (w \vee y \vee \neg z) \wedge (\neg x \vee y \vee z)$.
- $(\Phi' \mid x = \mathit{true}) = (w \vee y \vee \neg z) \wedge (y \vee z)$.

each clause has $\leq$ 3 literals

Exact Algorithms for 3-SAT

A recursive algorithm (divide and conquer). A 3-SAT formula Φ is either empty or the disjunction of a clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ and a 3-SAT formula Φ' with one fewer clause.

```
3-SAT ($\Phi$)
------------------------------------------
IF $\Phi$ is empty RETURN true.
/* Divide and Conquer */
IF 3-SAT ($\Phi \mid \ell_1 = true$) RETURN true.
IF 3-SAT ($\Phi \mid \ell_2 = true$) RETURN true.
IF 3-SAT ($\Phi \mid \ell_3 = true$) RETURN true.
RETURN false.
```

Since the depth of the recursion calls is n, then:

- $T(n) \leq 3T(n-1) + 3m$, the recursive 3-SAT algorithm above takes $O(m3^n)$ time, but...

Exact algorithms for 3-satisfiability

Key observation. The cases are not mutually exclusive. Every satisfiable assignment containing clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ must fall into one of 3 classes:

- ℓ_1 is *true*.
- ℓ_1 is *false*; ℓ_2 is *true*.
- ℓ_1 is *false*; ℓ_2 is *false*; ℓ_3 is *true*.

```
3-SAT (Φ)
IF Φ is empty RETURN true.
/* Divide and Conquer */
IF 3-SAT($\Phi \mid \ell_1 = true$) RETURN true.
IF 3-SAT($\Phi \mid \ell_1 = false, \ell_2 = true$) RETURN true.
IF 3-SAT($\Phi \mid \ell_1 = false, \ell_2 = false, \ell_3 = true$) RETURN true.
RETURN false.
```

Exact algorithms for 3-satisfiability

Theorem. The Divide and Conquer algorithm takes $O(m 1.84^n)$ time.

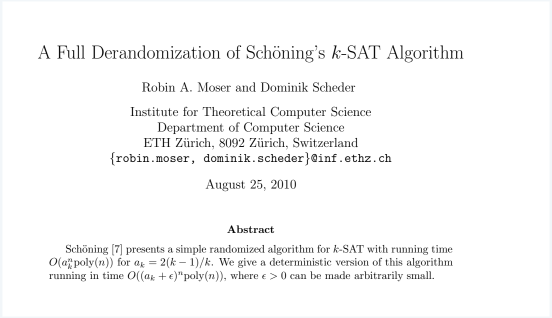
Pf. $T(n) \leq T(n-1) + T(n-2) + T(n-3) + O(m)$. ▪

largest root of $r^3 = r^2 + r + 1$

```
3-SAT (Φ)
________________________________________________
IF Φ is empty RETURN true.
/* Divide and Conquer */
IF 3-SAT(Φ | $\ell_1$ = true) RETURN true.
IF 3-SAT(Φ | $\ell_1$ = false, $\ell_2$ = true) RETURN true.
IF 3-SAT(Φ | $\ell_1$ = false, $\ell_2$ = false, $\ell_3$ = true) RETURN true.
RETURN false.
```

Exact algorithms for 3-satisfiability

Theorem. There exists a $O(1.33334^n)$ deterministic algorithm for 3-SAT.



A Full Derandomization of Schöning's k-SAT Algorithm

Robin A. Moser and Dominik Scheder

Institute for Theoretical Computer Science
Department of Computer Science
ETH Zürich, 8092 Zürich, Switzerland
{robin.moser, dominik.scheder}@inf.ethz.ch

August 25, 2010

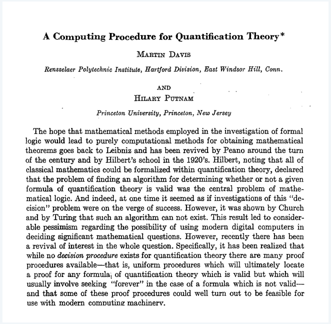
Abstract

Schöning [7] presents a simple randomized algorithm for k-SAT with running time $O(a_k^n \text{poly}(n))$ for $a_k = 2(k-1)/k$. We give a deterministic version of this algorithm running in time $O((a_k + \epsilon)^n \text{poly}(n))$, where $\epsilon > 0$ can be made arbitrarily small.

Exact algorithms for satisfiability

DPPL algorithm. Highly-effective backtracking procedure.

- Splitting rule: assign truth value to literal; solve both possibilities.
- Unit propagation: clause contains only a single unassigned literal.
- Pure literal elimination: if literal appears only negated or unnegated.



A Computing Procedure for Quantification Theory*

MARTIN DAVIS

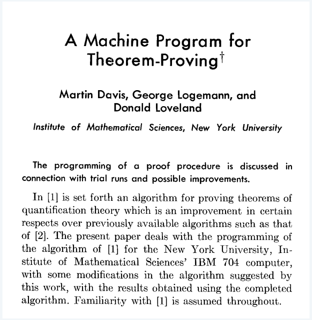
Rensselaer Polytechnic Institute, Hartford Division, East Windsor Hill, Conn.

AND

HILARY PUTNAM

Princeton University, Princeton, New Jersey

The hope that mathematical methods employed in the investigation of formal logic would lead to purely computational methods for obtaining mathematical theorems goes back to Leibniz and has been revived by Peano around the turn of the century and by Hilbert's school in the 1920's. Hilbert, noting that all of classical mathematics could be formalized within quantification theory, declared that the problem of finding an algorithm for determining whether or not a given formula of quantification theory is valid was the central problem of mathematical logic. And indeed, at one time it seemed as if investigations of this "decision" problem were on the verge of success. However, it was shown by Church and by Turing that such an algorithm can not exist. This result led to considerable pessimism regarding the possibility of using modern digital computers in deciding significant mathematical questions. However, recently there has been a revival of interest in the whole question. Specifically, it has been realized that while no *decision procedure* exists for quantification theory there are many proof procedures available—that is, uniform procedures which will ultimately locate a proof for any formula of quantification theory which is valid but which will usually involve seeking "forever" in the case of a formula which is not valid—and that some of these proof procedures could well turn out to be feasible for use with modern computing machinery.



A Machine Program for Theorem-Proving†

Martin Davis, George Logemann, and Donald Loveland

Institute of Mathematical Sciences, New York University

The programming of a proof procedure is discussed in connection with trial runs and possible improvements.

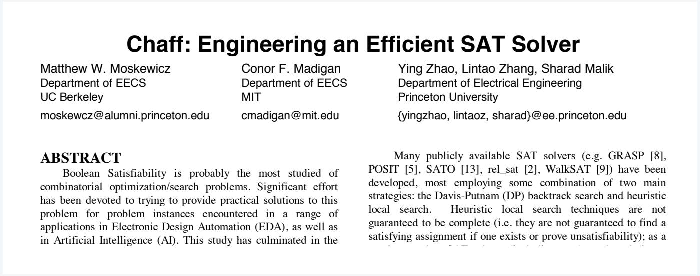
In [1] is set forth an algorithm for proving theorems of quantification theory which is an improvement in certain respects over previously available algorithms such as that of [2]. The present paper deals with the programming of the algorithm of [1] for the New York University, Institute of Mathematical Sciences' IBM 704 computer, with some modifications in the algorithm suggested by this work, with the results obtained using the completed algorithm. Familiarity with [1] is assumed throughout.

Exact algorithms for satisfiability

Chaff. State-of-the-art SAT solver.

- Solves real-world SAT instances with ~ 10K variable.

  Developed at Princeton by undergrads.



Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz
Department of EECS
UC Berkeley
moskewcz@alumni.princeton.edu

Conor F. Madigan
Department of EECS
MIT
cmadigan@mit.edu

Ying Zhao, Lintao Zhang, Sharad Malik
Department of Electrical Engineering
Princeton University
{yingzhao, lintaoz, sharad}@ee.princeton.edu

ABSTRACT

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in Artificial Intelligence (AI). This study has culminated in the

Many publicly available SAT solvers (e.g. GRASP [8], POSIT [5], SATO [13], rel_sat [2], WalkSAT [9]) have been developed, most employing some combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. Heuristic local search techniques are not guaranteed to be complete (i.e. they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability); as a

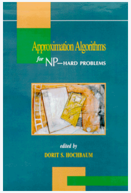
Approximation algorithms

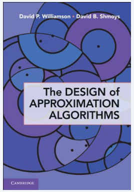
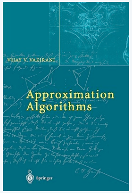
ρ-approximation algorithm.

- Runs in polynomial time.
- Applies to arbitrary instances of the problem.
- Guaranteed to find a solution within ratio ρ of true optimum.

Ex. Given a graph G, can find a vertex cover that uses $\leq 2\,OPT(G)$ vertices in $O(m+n)$ time.

Challenge. Need to prove a solution's value is close to optimum value, without even knowing what optimum value is!




David P. Williamson • David B. Shmoys
The DESIGN of APPROXIMATION ALGORITHMS
CAMBRIDGE


Degrees of Approximability

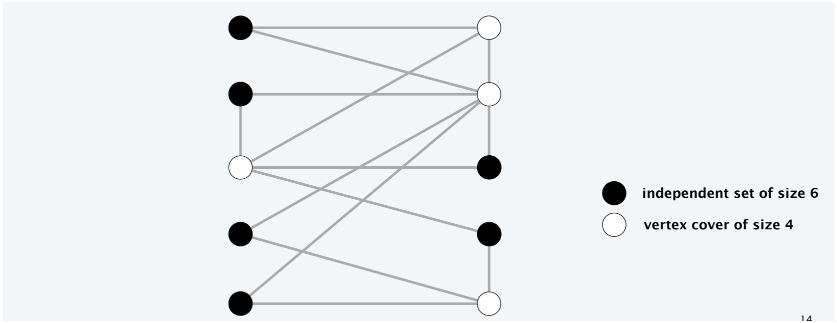
Assuming P $\neq$ NP, there is a difference between the NP-complete problems regarding how hard they are to approximate:

1. For some problems you can find a polynomial algorithm with approximation quotient $1 + \epsilon$, for every $\epsilon > 0$.
   Ex.: The Knapsack Problem.
2. Other problems can be approximated within a constant > 1 but not arbitrarily close to 1.
   Ex.: Vertex Cover
3. The are problems that cannot be approximated within any constant.
   Ex. Set Cover

Minimal Vertex Cover

Definition. Given a graph $G = (V, E)$ find a minimal $S \subseteq V$ which is a vertex cover.

Note. For each edge $(u, v) \in E$, either $u \in S$ or $v \in S$.



Minimal Vertex Cover: Greedy Algorithm

```
GREEDY-VERTEX-COVER($G$)
$S \leftarrow \varnothing$.
$E' \leftarrow E$.
WHILE ($E' \neq \varnothing$)
    Let $(u, v) \in E'$ be an arbitrary edge.
    $M \leftarrow M \cup \{(u, v)\}$.  ← $M$ is a matching
    $S \leftarrow S \cup \{u\} \cup \{v\}$.  ← every vertex cover must take at least one of these; we take both
    Delete from $E'$ all edges incident to either $u$ or $v$.
RETURN $S$.
```

Running time. Can be implemented in $O(m + n)$ time.

Minimal Vertex Cover: Greedy Algorithm/2

Theorem. Let S^* be a minimum vertex cover. Then, greedy algorithm computes a vertex cover S such that $|S| \leq 2|S^*|$.

Proof.

- S is a *vertex cover*: Delete an edge only after it is already covered.
- M is a *matching* (set of edges without common vertices): When (u, v) is added to M all edges incident to either u or v are deleted.
- $|M| \leq |S^*|$: S^* is a vertex cover and edges in M do not share vertices.
- $|S| = 2|M| \leq 2|S^*|$.

Minimal Vertex Cover: Greedy Algorithm/2

Theorem. Let S^* be a minimum vertex cover. Then, greedy algorithm computes a vertex cover S such that $|S| \leq 2|S^*|$.

Proof.

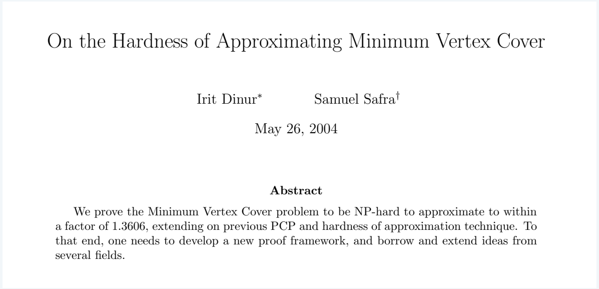
- S is a *vertex cover*: Delete an edge only after it is already covered.
- M is a *matching* (set of edges without common vertices): When (u, v) is added to M all edges incident to either u or v are deleted.
- $|M| \leq |S^*|$: S^* is a vertex cover and edges in M do not share vertices.
- $|S| = 2|M| \leq 2|S^*|$.

Corollary. Let M^* be a maximum matching, then, the greedy algorithm computes a matching M such that $|M| \geq 0.5|M^*|$.

Proof. $|M| = 0.5|S| \geq 0.5|S^*|$, since $|S^*| \geq |M^*|$, then, $|M| \geq 0.5|M^*|$.

Vertex cover inapproximability

Theorem. [Dinur–Safra 2004] If **P** ≠ **NP**, then no ρ-approximation for VERTEX-COVER for any $\rho < 1.3606$.



On the Hardness of Approximating Minimum Vertex Cover

Irit Dinur* Samuel Safra†

May 26, 2004

Abstract

We prove the Minimum Vertex Cover problem to be NP-hard to approximate to within a factor of 1.3606, extending on previous PCP and hardness of approximation technique. To that end, one needs to develop a new proof framework, and borrow and extend ideas from several fields.

Open research problem. Close the gap.

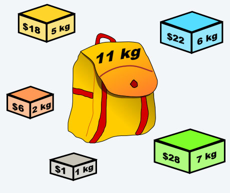
Conjecture. no ρ-approximation for VERTEX-COVER for any $\rho < 2$.

Knapsack Problem

Knapsack problem.

- Given n objects and a knapsack with weight limit W;
- Each object x_i has value $v_i > 0$ and weigh $w_i > 0$;
- Goal: fill the knapsack up to the weight limit so as to maximize the total value.

Ex. $\{1, 2, 5\}$ has value \$35 (and weight 10).
Ex. $\{3, 4\}$ has value \$40 (and weight 11).



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance (weight limit W = 11)

Knapsack Problem/2

Definition. Let $X = \{x_1, \ldots, x_n\}$ be a set where each object x_i has value $v_i > 0$ and weight $w_i > 0$, while the knaspack has weight limit W. Find a subset $S \subseteq X$ such that:

- $\sum_{x_i \in S} w_i \leq W$, and
- $\sum_{x_i \in S} v_i$ is maximal.

Knapsack Problem – The Decisional Version

Definition as a decision problem. Let $X = \{x_1, \ldots, x_n\}$ be a set where each object x_i has value $v_i > 0$ and weigh $w_i > 0$, while the knaspack has weight limit W. Fixed an integer V, is there a subset $S \subseteq X$ such that:

- $\sum_{x_i \in S} w_i \leq W$, and
- $\sum_{x_i \in S} v_i \geq V$?

Knapsack Problem – The Decisional Version

Definition as a decision problem. Let $X = \{x_1, \ldots, x_n\}$ be a set where each object x_i has value $v_i > 0$ and weigh $w_i > 0$, while the knaspack has weight limit W. Fixed an integer V, is there a subset $S \subseteq X$ such that:

- $\sum_{x_i \in S} w_i \leq W$, and
- $\sum_{x_i \in S} v_i \geq V$?

Theorem. Knapsack is an NP-complete problem.

Dynamic Programming

Main Characterisation.

- The basic idea is drawn from the intuition behind divide and conquer and is essentially the opposite of the greedy strategy;
- Dynamic Programming explores the space of all possible solutions by decomposing into a series of subproblems, and
- The solution to the original problem can be obtained easily once we know the solutions to all subproblems.

Knapsack as a Scheduling Problem

The Knapsack problem is occurring in many scheduling problems.

Scheduling Problem.

- We have a single machine that can process n jobs;
- The machine is free for the period between time 0 and time W, for some number W;
- Job i requires time w_i to be processed.
- Goal: Choose a job scheduling so as to keep the machine as busy as possible up to the “cut-off” W.

Knapsack Vs. Scheduling. The scheduling problem is a Knapsack where $v_i = w_i$ and it is also known as the Subset Sum Problem.

Subset Sum Problem – Dynamic Programming

`OPT(i,w)`: denotes the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ with maximum allowed weight w, that is:

- $0 \leq w \leq W$, and
- $S \subseteq \{1, \ldots, i\}$ such that $\sum_{j \in S} w_j \leq w$, then
- $\texttt{OPT(i,w)} = \max_S \left(\sum_{j \in S} w_j \right)$.

Subset Sum Problem – Dynamic Programming

`OPT(i,w)`: denotes the value of the optimal solution using a subset of the items $\{1,\ldots,i\}$ with maximum allowed weight w, that is:

- $0 \leq w \leq W$, and
- $S \subseteq \{1,\ldots,i\}$ such that $\sum_{j\in S} w_j \leq w$, then
- $\mathtt{OPT(i,w)} = \max_S \left(\sum_{j\in S} w_j\right)$.

`OPT(n,W)` is the quantity we are looking for!

Subset Sum Problem – Bellman Equation

Let $\mathcal{O}$ denote an optimum solution for the problem, we then have 3 cases:

1. If $w_n > W$, then $\mathtt{OPT(n, W)} = \mathtt{OPT(n-1, W)}$, otherwise
2. If $n \notin \mathcal{O}$, then, $\mathtt{OPT(n, W)} = \mathtt{OPT(n-1, W)}$;
3. If $n \in \mathcal{O}$, then, $\mathtt{OPT(n, W)} = \mathtt{w_n} + \mathtt{OPT(n-1, W-w_n)}$.

Subset Sum Problem – Bellman Equation

Let $\mathcal{O}$ denote an optimum solution for the problem, we then have 3 cases:

1. If $w_n > W$, then $\texttt{OPT}(\texttt{n}, \texttt{W}) = \texttt{OPT}(\texttt{n} - 1, \texttt{W})$, otherwise
2. If $n \notin \mathcal{O}$, then, $\texttt{OPT}(\texttt{n}, \texttt{W}) = \texttt{OPT}(\texttt{n} - 1, \texttt{W})$;
3. If $n \in \mathcal{O}$, then, $\texttt{OPT}(\texttt{n}, \texttt{W}) = \texttt{w}_\texttt{n} + \texttt{OPT}(\texttt{n} - 1, \texttt{W} - \texttt{w}_\texttt{n})$.

Extending the above argument to the subproblems $\{1, \ldots, i\}$ we have the following recurrence (Bellman equation):

$$\begin{gathered}\text{If } w_i > w,\ \text{then, } \texttt{OPT}(\texttt{i}, \texttt{w}) = \texttt{OPT}(\texttt{i} - 1, \texttt{w}).\ \text{Otherwise,} \\ \texttt{OPT}(\texttt{i}, \texttt{w}) = \max\left\{\texttt{OPT}(\texttt{i} - 1, \texttt{w}), w_i + \texttt{OPT}(\texttt{i} - 1, \texttt{w} - \texttt{w}_\texttt{i})\right\}.\end{gathered} \tag{1}$$

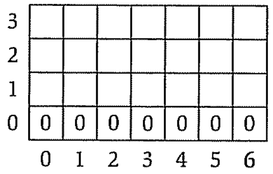
Subset Sum Problem – Dynamic Programming Algorithm

Algorithm that builds up a table of all `OPT(i,w)` values, $i = 1, \dots, n$ and $0 \leq w \leq W$, while computing each of them at most once.

```
SUBSET-SUM($n, W, w_1, \dots, w_n$)
Array OPT[$0..n, 0..W$];
for $w = 0, 1, \dots W$ do
    OPT[$0, w$] $= 0$;                    /* set to 0 the first raw of the table */
for $i = 1, \dots, n$ do
    for $w = 0, \dots, W$ do
        if $w_i > w$ then OPT[$i, w$] $=$ OPT[$i-1, w$];
        else
            OPT[$i, w$] $= \max\{$OPT[$i-1, w$]$, w_i +$ OPT[$i-1, w-w_i$]$\}$
return OPT[$n, W$]
```

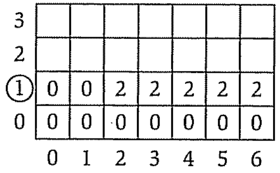
Subset Sum – Running Example

Knapsack size $W = 6$, items $w_1 = 2$, $w_2 = 2$, $w_3 = 3$



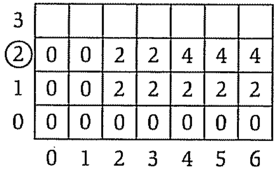
3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values



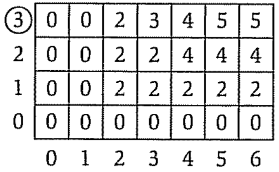
3							
2							
①	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 1$



3							
②	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 2$



③	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 3$

Subset Sum Problem: Running Time

Running Time. The Dynamic Programming algorithm solves the Subset Sum problem with n items and weight limit W in $O(nW)$ time and $O(nW)$ space.

- It computes each of the values $\texttt{OPT}[i, w]$ in $O(1)$ time using the previous values;
- Since there are nW values, the total running time is $O(nW)$.

Subset Sum Problem: Running Time

Running Time. The Dynamic Programming algorithm solves the Subset Sum problem with n items and weight limit W in $O(nW)$ time and $O(nW)$ space.

- It computes each of the values $\texttt{OPT}[i, w]$ in $O(1)$ time using the previous values;
- Since there are nW values, the total running time is $O(nW)$.

The DP Algorithm is linear in n, BUT Exponential in the bit representation of W!!!

- Pseudo-polynom Algorithm. It runs in time polynomial in the magnitude of the input numbers, but not polynomial in the size of their representation.

Subset Sum: Find the Solution

- After computing the matrix for optimal values, we can trace back to find a solution;
- We use the following condition:
  i is selected by $\texttt{OPT}[i, w]$ iff $\texttt{OPT}[i, w] > \texttt{OPT}[i-1, w]$
- Let $\mathcal{O}$ be the optimal solution. We call the following recursive procedure with:
  $i = n,\ w = W,\ \mathcal{O} = \emptyset$, while the matrix OPT is already computed.

```
FIND-SOLUTION($i, w$)
if $i$=$0$ then stop;
if $\texttt{OPT}[i, w] > \texttt{OPT}[i-1, w]$ then
    $\mathcal{O} = \mathcal{O} \cup \{i\}$;
    FIND-SOLUTION($i-1, w-w_i$)
else
    FIND-SOLUTION($i-1, w$)
```

Knapsack Problem/2

Definition. Let $X = \{x_1, \ldots, x_n\}$ be a set where each object x_i has value $v_i > 0$ and weigh $w_i > 0$, while the knaspack has weight limit W. Find a subset $S \subseteq X$ such that:

- $\sum_{x_i \in S} w_i \leq W$, and
- $\sum_{x_i \in S} v_i$ is maximal.

Knapsack Problem – Dynamic Programming

`OPT(i,w)`: denotes the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ with maximum allowed weight w, that maximizes the total value, that is:

- $0 \leq w \leq W$, and
- $S \subseteq \{1, \ldots, i\}$ such that $\sum_{j \in S} w_j \leq w$, then
- $\texttt{OPT}(\texttt{i}, \texttt{w}) = \max_S \left(\sum_{j \in S} v_j \right)$.

Knapsack Problem – Dynamic Programming

`OPT(i,w)`: denotes the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ with maximum allowed weight w, that maximizes the total value, that is:

- $0 \leq w \leq W$, and
- $S \subseteq \{1, \ldots, i\}$ such that $\sum_{j \in S} w_j \leq w$, then
- $\mathtt{OPT(i,w)} = \max_{S} \left(\sum_{j \in S} v_j \right)$.

`OPT(n,W)` is the quantity we are looking for!

Knapsack Problem – Dynamic Programming/2

Let $\mathcal{O}$ denote an optimum solution for the problem, then

- If $w_n > W$, then $\texttt{OPT(n, W)} = \texttt{OPT(n} - \texttt{1, W)}$, otherwise
- If $n \notin \mathcal{O}$, then, $\texttt{OPT(n, W)} = \texttt{OPT(n} - \texttt{1, W)}$;
- If $n \in \mathcal{O}$, then, $\texttt{OPT(n, W)} = \texttt{v}_\texttt{n} + \texttt{OPT(n} - \texttt{1, W} - \texttt{w}_\texttt{n}\texttt{)}$.

Extending the above argument to the subproblems $\{1, \ldots, i\}$ we have the following recurrence (Bellman equation):

$$\text{If } w_i > w, \text{ then, } \texttt{OPT(i, w)} = \texttt{OPT(i} - \texttt{1, w)}. \text{ Otherwise,}$$
$$\texttt{OPT(i, w)} = \max\left(\texttt{OPT(i} - \texttt{1, w)}, v_i + \texttt{OPT(i} - \texttt{1, w} - \texttt{w}_\texttt{i}\texttt{)}\right). \qquad (2)$$

Knapsack Problem – Dynamic Programming – The Algorithm

Algorithm that builds up a table of all `OPT(i,w)` values, $i = 1, \ldots, n$ and $0 \leq w \leq W$, while computing each of them at most once.

```
Knapsack($n, W, w_1, \ldots, w_n, v_1, \ldots, v_n$)
Array OPT[$0..n, 0..W$];
for $w = 0, 1, \ldots W$ do
    OPT[$0, w$] = 0
for $i = 1, \ldots, n$ do
    for $w = 0, \ldots, W$ do
        if $w_i > w$ then OPT[$i, w$] = OPT[$i - 1, w$];
        else
            OPT[$i, w$] = max{OPT[$i - 1, w$], $v_i$ + OPT[$i - 1, w - w_i$]}
return OPT[$n, W$]
```

Knapsack Problem – Dynamic Programming – The Algorithm

Algorithm that builds up a table of all `OPT(i,w)` values, $i = 1, \dots, n$ and $0 \le w \le W$, while computing each of them at most once.

```
Knapsack($n, W, w_1, \dots, w_n, v_1, \dots, v_n$)
Array OPT[$0..n, 0..W$];
for $w = 0, 1, \dots W$ do
    OPT[$0, w$] = 0
for $i = 1, \dots, n$ do
    for $w = 0, \dots, W$ do
        if $w_i > w$ then OPT[$i, w$] = OPT[$i - 1, w$];
        else
            OPT[$i, w$] = max{OPT[$i - 1, w$], $v_i$ + OPT[$i - 1, w - w_i$]}
return OPT[$n, W$]
```

Running Time: $O(nW)$.

Notes on the Complexity of Knapsack

Remarks.

- The DP algorithm has complexity $O(nW)$;
- The complexity depends both on the size of the input set, n, and on the weight W;
- If $W \in O(n^c)$, for some constant c, then Knapsack can be solved by DP in polynomial time, $O(n^{c+1})$;
- If $W \in O(2^n)$, then Knapsack is solved by DP in exponential time, $O(n2^n)$

Notes on the Complexity of Knapsack

Remarks.

- The DP algorithm has complexity $O(nW)$;
- The complexity depends both on the size of the input set, n, and on the weight W;
- If $W \in O(n^c)$, for some constant c, then Knapsack can be solved by DP in polynomial time, $O(n^{c+1})$;
- If $W \in O(2^n)$, then Knapsack is solved by DP in exponential time, $O(n2^n)$

There is no hope to solve Knaspack in polynomial time for all possible input instances, unless P = NP.

Theorem. The Knaspack problem is NP-complete.

Pseudo-Polynomial Algorithms

Definition. An algorithm A is pseudo-polynomial if the time complexity is polynomial in the size of the input, n, and the magnitude of M, the greatest number in the input, but not in its representation.

- For example, the DP algorithms solving `SUBSET-SUM` and `KNAPSACK` are pseudo-polynomial algorithms.

Pseudo-Polynomial Algorithms

Definition. An algorithm A is pseudo-polynomial if the time complexity is polynomial in the size of the input, n, and the magnitude of M, the greatest number in the input, but not in its representation.

- For example, the DP algorithms solving `SUBSET-SUM` and `KNAPSACK` are pseudo-polynomial algorithms.

Theorem. Unless $\mathrm{P} = \mathrm{NP}$, there are strongly NP-Complete problems, i.e., NP-Complete problems which cannot be solved by pseudo-polynomial algorithms.

Knapsack problem: dynamic programming II

Def. $OPT(i, v)$ = min weight of a knapsack for which we can obtain a solution of value $\geq v$ using a subset of items $1,..., i$.

Note. Optimal value is the largest value v such that $OPT(n, v) \leq W$.

Case 1. OPT does not select item i.

- OPT selects best of $1, \ldots, i-1$ that achieves value $\geq v$.

Case 2. OPT selects item i.

- Consumes weight w_i, need to achieve value $\geq v - v_i$.
- OPT selects best of $1, \ldots, i-1$ that achieves value $\geq v - v_i$.

$$OPT(i,v) = \begin{cases} 0 & \text{if } v \leq 0 \\ \infty & \text{if } i = 0 \text{ and } \ v > 0 \\ \min\{OPT(i-1,v),\ w_i + OPT(i-1, v-v_i)\} & \text{otherwise} \end{cases}$$

Knapsack Problem – Dynamic Programming II

Algorithm that builds up a table of all OPT(i,v) values, $i = 1, \ldots, n$ and $0 \leq v \leq V$, with $V = \Sigma_{i=1}^{n} v_i$.

```
Knapsack-DP-II($n, V, w_1, \ldots, w_n, v_1, \ldots, v_n$)
Array OPT[$0..n, 0..V$];
for $i = 0, 1, \ldots n$ do
    OPT[$i, 0$] $= 0$;                /* set to 0 the first column of the table */
for $i = 1, \ldots, n$ do
    for $v = 1, \ldots, \Sigma_{j=1}^{i} v_j$ do
        if $v > \Sigma_{j=1}^{i-1} v_j$ then OPT[$i, v$] $= w_i +$ OPT[$i-1, v - v_i$];
        else
            OPT[$i, v$] $= \min\{$OPT[$i-1, v$], $w_i +$ OPT[$i-1, \max\{0, v - v_i\}$]$\}$
return The max $V$ such that OPT[$n, V$] $\leq W$
```

Knapsack problem: dynamic programming II

Theorem. Dynamic programming algorithm II computes the optimal value in $O(n^2\, v_{\max})$ time, where $v_{\max}$ is the maximum of any value.

Pf.

- The optimal value $V^* \leq n\, v_{\max}$.
- There is one subproblem for each item and for each value $v \leq V^*$.
- It takes $O(1)$ time per subproblem. ▪

Remark 1. Not polynomial in input size!

Remark 2. Polynomial time if values are small integers.

Knapsack problem: poly-time approximation scheme

Intuition for approximation algorithm.

- Round all values up to lie in smaller range.
- Run dynamic programming algorithm II on rounded/scaled instance.
- Return optimal items in rounded instance.

item	value	weight
1	934221	1
2	5956342	2
3	17810013	5
4	21217800	6
5	27343199	7

original instance (W = 11)

item	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

rounded instance (W = 11)

Arbitrarily Good Approximations: The Knapsack Problem

- The Knapsack problem is an NP-complete problem for which it is possible to design a polynomial-time algorithm providing a very strong approximation.
- We show that there exists an algorithm that takes as input an extra parameter ϵ, the desired precision, such that:
  - It will find a subset S whose total weight does not exceed W, and
  - Value $\sum_{i \in S} v_i$ at most a $(1 + \epsilon)$ factor below the maximum possible;
  - Running in polynomial time for a fixed choice of $\epsilon > 0$;
  - However, the dependence on ϵ will not be polynomial, indeed the running time will be $O(\frac{n^3}{\epsilon})$;
  - For smaller and smaller values of ϵ, the running time gets larger and larger.

Arbitrarily Good Approximations: The Knapsack Problem/2

Ingredients.

- We use the variation of the presented DP algorithm that runs in time $O(n^2 v_{max})$, where $v_{max} = \max_i v_i$ to design a polynomial-time approximation algorithm.
- $0 \leq \epsilon \leq 1$, precision (we assume that ϵ^{-1} is integer);
- $b = \dfrac{\epsilon \cdot v_{max}}{2n}$, scaling factor;
- $\overline{v_i} = \left\lceil \dfrac{v_i}{b} \right\rceil \cdot b$; rounded values ($v_i \leq \overline{v_i} \leq v_i + b$);
- $\hat{v}_i = \dfrac{\overline{v_i}}{b} = \left\lceil \dfrac{v_i}{b} \right\rceil$; scaled values.

Property/1. The Knapsack Problem with rounded values $\overline{v_i}$ or with scaled values $\hat{v}_i$ has the same set of optimum solutions.

Arbitrarily Good Approximations: The Knapsack Problem/2

Approximated Algorithm.

```
Knapsack-Approx($n, W, w_1, \ldots, w_n, v_1, \ldots, v_n, \epsilon$)
$b = \frac{\epsilon \cdot v_{max}}{2n}$;
Solve the Knapsack problem with values $\hat{v}_i = \left\lceil \frac{v_i}{b} \right\rceil$ and in $O(n^2 \hat{v}_{max})$;
return the set $S_x$ of items found.
```

Knapsack-Approx – Analysis

Property/2. Knapsack-Approx runs in $O(\frac{n^3}{\epsilon})$, i.e., in polynomial time for any fixed $\epsilon > 0$.

Proof. $\hat{v}_{max} = \left\lceil \frac{v_{max}}{b} \right\rceil = \left\lceil \frac{v_{max} \cdot 2n}{\epsilon \cdot v_{max}} \right\rceil = \frac{2n}{\epsilon}$. Thus, $O(n^2 \hat{v}_{max}) = O(\frac{n^3}{\epsilon})$.

Knapsack-Approx – Analysis/2

Property/3. If S_x is a solution found by the Knapsack-Approx algorithm and S^* is any other possible solution then, $(1+\epsilon)\sum_{i\in S_x} v_i \geq \sum_{i\in S^*} v_i$.

Proof. By Property/1 S_x is also an optimal solution with values $\overline{v_i}$, then: $\sum_{i\in S^*} \overline{v_i} \leq \sum_{i\in S_x} \overline{v_i}$, and since $v_i \leq \overline{v_i} \leq v_i + b$, then:

$$\sum_{i\in S^*} v_i \leq \sum_{i\in S^*} \overline{v_i} \leq \sum_{i\in S_x} \overline{v_i} \leq \sum_{i\in S_x} (v_i + b) \leq nb + \sum_{i\in S_x} v_i \tag{3}$$

Note that

- $\overline{v}_{max} = v_{max}$ when ϵ^{-} is an integer, and
- by assumption, each item alone fits in the knapsack (i.e., $w_i \leq W$, for all i).

$$\sum_{i\in S_x} \overline{v_i} \geq \overline{v}_{max} = v_{max} = \frac{2n\cdot b}{\epsilon} \tag{4}$$

Knapsack-Approx – Analysis/3

Proof continued.

Form (3) and (4), it follows:

$$\sum_{i\in S_x} v_i \geq \sum_{i\in S_x} \overline{v_i} - nb \geq \frac{2n\cdot b}{\epsilon} - nb = nb\Big(\frac{2}{\epsilon} - 1\Big) \tag{5}$$

and thus, when $\epsilon \leq 1$:

$$nb \leq \frac{\sum_{i\in S_x} v_i}{\Big(\frac{2}{\epsilon} - 1\Big)} \leq \frac{\sum_{i\in S_x} v_i}{\Big(\frac{1}{\epsilon}\Big)} = \epsilon \sum_{i\in S_x} v_i \tag{6}$$

From (3) and (6), we finally get:

$$\sum_{i\in S^*} v_i \leq nb + \sum_{i\in S_x} v_i \leq \epsilon \sum_{i\in S_x} v_i + \sum_{i\in S_x} v_i = (1+\epsilon)\sum_{i\in S_x} v_i . \tag{7}$$

Thank You!