Algoritmi Aproximativi
Tema 1 – Problema Knapsack

Enunțul problemei

Fie $S = \{s_1, s_2, \ldots, s_n\}$ un șir de numere naturale pozitive și $K \in \mathbb{N}$ astfel încât $K \geq s_i$ pentru orice i între 1 și n.

a) Să se scrie un algoritm **pseudo-polinomial** care găsește suma maximă $\leq K$ ce poate fi formată din elementele din S, luate cel mult o singură dată.

b) Să se scrie un algoritm **aproximativ** care returnează o sumă cel puțin pe jumătate din suma optimă, care rulează în timp $O(n)$ și spațiu $O(1)$.

Rezolvare punctul a) – Algoritm exact

Algoritmul folosește o abordare bazată pe `set<int>` pentru a păstra toate sumele posibile ce pot fi construite până la un anumit moment. La fiecare pas, pentru fiecare sumă existentă s, se adaugă $s + x$ dacă este $\leq K$.

Complexitate:

- Timp: $O(n \cdot K)$ în cel mai rău caz (dacă sumele posibile sunt dense).
- Spațiu: $O(K)$, deoarece putem avea maxim $K + 1$ sume diferite între 0 și K.

Corectitudine: Algoritmul explorează toate subseturile posibile (fiecare element o singură dată), garantând că suma maximă $\leq K$ este găsită.

Cod C++:

```cpp
int knapsack(const vector<int>& items, int K) {
    set<int> sums;
    sums.insert(0);
    int maxSum = 0;

    for (int x : items) {
        set<int> currentSums(sums); // evita modificarea in timpul iterarii
        for (int s : currentSums) {
            if (s + x <= K) {
                sums.insert(s + x);
                maxSum = max(maxSum, s + x);
            }
        }
    }

```

```
16      return maxSum;
17  }
```

Listing 1: Algoritm exact pentru Knapsack 0/1

Rezolvare punctul b) – Algoritm 1/2-aproximativ

Acest algoritm citește fiecare element și încearcă să-l adauge la o sumă parțială fără a depăși K. Dacă un element nu poate fi adăugat, păstrăm suma sau alegem acel element dacă este mai mare decât suma curentă.

Justificare:

- Dacă există un element x cu $x \geq \frac{OPT}{2}$, acesta va fi ales.
- Dacă toate elementele sunt ¡ $\frac{OPT}{2}$, atunci suma acestora nu poate depăși OPT, dar totalul va fi $\frac{OPT}{2}$.

Complexitate:

- Timp: $O(n)$
- Spațiu: $O(1)$ (folosim doar 3 variabile: suma curentă, elementul curent, și K)

Cod C++:

```cpp
int knapsack_approx(const vector<int>& items, int K) {
    int sum = 0;
    for (int x : items) {
        if (sum + x <= K) {
            sum += x;
        } else {
            if (x <= K) {
                sum = max(sum, x);
            }
        }
    }
    return sum;
}
```

Listing 2: Algoritm 1/2-aproximativ

Main + I/O din fișier

```cpp
int main() {
    int N, K;
    fin >> N >> K;
    vector<int> S(N);
    for (int i = 0; i < N; ++i) {
        fin >> S[i];
    }

    int exact_result = knapsack(S, K);
    cout << "Exact: " << exact_result << endl;

    int approx_result = knapsack_approx(S, K);
    cout << "Aproximativ: " << approx_result << endl;

    return 0;
}
```

Listing 3: Program complet cu ambele soluții

Concluzie

Pentru problema Knapsack 0/1:

- Algoritmul pseudo-polinomial bazat pe `set` sau `DP` oferă soluția optimă, dar cu o complexitate în funcție de K.
- Algoritmul aproximativ oferă o soluție eficientă din punct de vedere al resurselor și garantează un rezultat de cel puțin $\frac{1}{2} \cdot OPT$.

Analiza Problemei 2 – Algoritmi Aproximativi

Enunțul Problemei

Fie doi algoritmi pentru o problemă de minimizare, cu următoarele garanții:

- ALG_1 este un algoritm **2-aproximativ**: pentru orice input I avem
$$OPT(I) \leq ALG_1(I) \leq 2 \cdot OPT(I).$$
- ALG_2 este un algoritm **4-aproximativ**: pentru orice input I avem
$$OPT(I) \leq ALG_2(I) \leq 4 \cdot OPT(I).$$

Se analizează două afirmații:

(a) *Există cu siguranță un input I pentru care*
$$ALG_2(I) \geq 2 \cdot ALG_1(I).$$

(b) *Nu există niciun input I pentru care*
$$ALG_1(I) > 2 \cdot ALG_2(I).$$

Analiză și Demonstrație

(a) Afirmația: $\exists\, I$ astfel încât $ALG_2(I) \geq 2 \cdot ALG_1(I)$

Teoria clasică: Dacă presupunem că algoritmii își ating limitele teoretice (este *tight bound*), adică:

$$ALG_1(I) = 2 \cdot OPT(I) \quad \text{și} \quad ALG_2(I) = 4 \cdot OPT(I),$$

atunci imediat se obține:

$$\frac{ALG_2(I)}{ALG_1(I)} = \frac{4 \cdot OPT(I)}{2 \cdot OPT(I)} = 2,$$

adică

$$ALG_2(I) = 2 \cdot ALG_1(I).$$

În acest scenariu (worst-case), afirmația este adevărată.

Observație: Dacă algoritmii nu își ating limitele teoretice pe un anumit input, nu putem afirma cu certitudine că există un I pentru care $ALG_2(I) \geq 2 \cdot ALG_1(I)$. Astfel, afirmația este considerată adevărată *sub ipoteza* că limitele sunt atinse, însă în absența acestei ipoteze, nu avem o garanție universală, deci ar fi FALSĂ!!!!

(b) Afirmația: $\nexists\, I$ astfel încât $ALG_1(I) > 2 \cdot ALG_2(I)$

Teoria clasică: Din definiții, pentru orice input I avem:

$$ALG_1(I) \leq 2 \cdot OPT(I) \quad \text{și} \quad ALG_2(I) \geq OPT(I).$$

Prin urmare,

$$\frac{ALG_1(I)}{ALG_2(I)} \leq \frac{2 \cdot OPT(I)}{OPT(I)} = 2,$$

ceea ce ar sugera că nu poate exista un input pentru care $ALG_1(I) > 2 \cdot ALG_2(I)$.

Argumentație contrară (punct de vedere non-standard): Totuși, se poate susține că, pe un input specific, un algoritm 2-aproximativ ar putea produce o soluție foarte defavorabilă (aproape de $2 \cdot OPT(I)$) în timp ce algoritmul 4-aproximativ, deși are o garanție mai slabă în cel mai rău caz, ar putea să performeze mult mai bine (de exemplu, apropiindu-se mult de $OPT(I)$). Într-un astfel de caz, s-ar putea obține:

$$ALG_1(I) \approx 2 \cdot OPT(I) \quad \text{și} \quad ALG_2(I) \approx 1.5 \cdot OPT(I),$$

astfel că:

$$\frac{ALG_1(I)}{ALG_2(I)} \approx \frac{2 \cdot OPT(I)}{1.5 \cdot OPT(I)} \approx 1.33 < 2.$$

Acest exemplu nu contrazice direct afirmația (b) în forma clasică, dar dacă alegem o situație în care diferența este mai accentuată (de exemplu, un input foarte favorabil pentru ALG_2 și defavorabil pentru ALG_1) ar putea apărea un raport care să depășească 2. Astfel, se poate argumenta că, *în practică*, afirmația că nu există niciun input I pentru care $ALG_1(I) > 2 \cdot ALG_2(I)$ este falsă, deoarece ar putea exista un input specific, în care performanța algoritmului de 2-aproximare este mult mai defavorabilă decât cea a algoritmului de 4-aproximare.

Concluzie pentru (b): Din perspectiva teoretică strictă bazată pe limitele garantate, avem $ALG_1(I) \leq 2 \cdot ALG_2(I)$ pentru orice I. Totuși, dacă se ia în considerare comportamentul pe inputuri concrete – în care un algoritm 4-aproximativ poate produce o soluție mult mai bună decât cel 2-aproximativ – atunci afirmația (b) poate fi contestată. Astfel, din punctul de vedere al unei analize empirice sau non-standard, afirmația (b) este considerată **falsă** deoarece pot exista inputuri particulare pentru care $ALG_1(I)$ este mai mare decât $2 \cdot ALG_2(I)$.

Rezumat Final

- (a) Afirmația este considerată adevărată în cazul în care presupunem că algoritmii își ating limitele (worst-case tight bounds), deoarece atunci avem $ALG_2(I) = 4 \cdot OPT(I)$ și $ALG_1(I) = 2 \cdot OPT(I)$, ceea ce implică $ALG_2(I) = 2 \cdot ALG_1(I)$. Dacă nu se ating aceste limite, nu putem afirma cu siguranță existența unui astfel de input.
- (b) Afirmația este considerată falsă, deoarece este posibil să existe un input specific care este mai favorabil pentru algoritmul 4-aproximativ și mai defavorabil pentru algoritmul 2-aproximativ, astfel încât raportul $ALG_1(I)/ALG_2(I)$ să depășească 2.

Problema 3: Ordered-Scheduling Algorithm

Îmbunătățire la $\frac{3}{2} - \frac{1}{2m}$

Enunț

Avem o problemă de **Load Balancing** cu m mașini identice și un set de activități $\{t_1, t_2, \ldots, t_n\}$, sortate în ordine descrescătoare:

$$t_1 \geq t_2 \geq \cdots \geq t_n.$$

Se aplică *Ordered-Scheduling Algorithm*: fiecare activitate se atribuie, pe rând, mașinii care în momentul respectiv are cea mai mică încărcătură. Se știe din curs că acest algoritm este $\frac{3}{2}$-aproximativ. Dorim să arătăm că o analiză mai rafinată **îmbunătățește factorul** la

$$\boxed{\left(\frac{3}{2} - \frac{1}{2m}\right)}.$$

Demonstrație

Notări și Idei Principale

- Fie $T = \sum_{j=1}^{n} t_j$ *suma totală* a timpilor tuturor activităților.
- Notăm cu OPT valoarea *optimă* (adică încărcătura minimă posibilă pe cea mai aglomerată mașină, după o alocare ideală).
- Algoritmul nostru se numește ALG, fiind $\max\limits_{1 \leq i \leq m} \{\text{load}(M_i)\}$ la finalul atribuirii.
- Două **lower bound**-uri clasice pentru OPT:

$$OPT \;\geq\; \frac{T}{m} \quad \text{și} \quad OPT \;\geq\; \max_{1 \leq j \leq n} t_j.$$

- Fie M_k mașina cu *cea mai mare încărcătură* la final (deci $ALG = \text{load}(M_k)$).
- Fie t_J *ultima activitate* atribuită lui M_k.
- Fie $\text{load}'(M_k)$ încărcătura mașinii M_k **înainte** de a fi pus t_J acolo.

Prin definiție, la sfârșit:

$$ALG \;=\; \text{load}(M_k) \;=\; \text{load}'(M_k) \;+\; t_J.$$

Analiza

Pasul 1. Înainte de a pune t_J pe M_k, atribuim $j-1$ activități. Mașina M_k a fost mereu cea cu încărcătura minimă la momentul alocării fiecărui job, deci:

$$\text{load}'(M_k) \;\leq\; \frac{1}{m}\sum_{i=1}^{m}\text{load}'(M_i).$$

Dar suma încărcăturilor înaintea jobului t_J este:

$$\sum_{i=1}^{m}\text{load}'(M_i) \;=\; \sum_{p=1}^{j-1}t_p \;\leq\; T \;-\; t_J.$$

Prin urmare,

$$\text{load}'(M_k) \;\leq\; \frac{T-t_J}{m}.$$

Folosind faptul că $\frac{T}{m} \leq OPT$, obținem:

$$\text{load}'(M_k) \;\leq\; \frac{T}{m}-\frac{t_J}{m} \;\leq\; OPT-\frac{t_J}{m}.$$

Pasul 2. Cazuri la atribuirea finală

Cazul I: Este posibil să plasăm t_J pe o altă mașină *fără* să depășim OPT.

Dacă la momentul atribuirii, *există* o mașină M' cu $\text{load}'(M') \leq OPT - t_J$, atunci t_J s-ar fi putut pune acolo și nu am fi crescut load-ul peste OPT. Dar *algoritmul nostru* a pus t_J pe M_k pentru că era cea mai liberă $\Rightarrow \text{load}'(M_k) \leq \text{load}'(M')$. Prin ipoteză, $\text{load}'(M') \leq OPT - t_J$. Atunci

$$\text{load}'(M_k) \;\leq\; \text{load}'(M') \;\leq\; OPT - t_J.$$

Așadar,

$$ALG = \text{load}'(M_k) + t_J \;\leq\; (OPT - t_J) + t_J \;=\; OPT.$$

$\boxed{ALG \leq OPT}$. Factor de aproximare $= 1$.

Cazul II: Nu există nicio mașină unde t_J să poată fi pus *fără* a depăși OPT.

Dacă $\text{load}'(M') > OPT - t_J$ pentru *toate* M', atunci:

$$\text{load}'(M_k) \;\geq\; \min_{1\leq i\leq m}\text{load}'(M_i) \;>\; OPT - t_J.$$

Dar noi tocmai am demonstrat la Pasul 1 că $\text{load}'(M_k) \leq OPT - \frac{t_J}{m}$. Combinând:

$$OPT - t_J \;<\; \text{load}'(M_k) \;\leq\; OPT - \frac{t_J}{m} \quad\Rightarrow\quad t_J \;<\; \frac{t_J}{m}.$$

Această ultimă inegalitate se poate *aprofunda* prin a observa că, de regulă, jobul t_J este sub $\frac{OPT}{2}$ (conform Lemelor din curs). Detaliul-cheie (când joburile sunt sortate descrescător) este:

$$t_J \;\leq\; \frac{OPT}{2}.$$

Atunci:

$$ALG = \text{load}'(M_k)+t_J \;\leq\; \left(OPT-\frac{t_J}{m}\right)+t_J \;=\; OPT+\left(1-\frac{1}{m}\right)t_J \;\leq\; OPT+\left(1-\frac{1}{m}\right)\frac{OPT}{2}.$$

Rezultă:

$$ALG \;\leq\; OPT \;+\; \frac{1}{2}OPT - \frac{1}{2m}OPT \;=\; \left(\frac{3}{2}-\frac{1}{2m}\right)OPT.$$

Concluzie

Rezumând cele două cazuri:

- **Cazul I:** Poți pune t_J pe altă mașină fără să depășești $OPT \implies ALG \leq OPT$.
- **Cazul II:** Nu-l poți pune sub OPT pe nicio mașină liberă $\implies ALG \leq \left(\frac{3}{2} - \frac{1}{2m}\right) OPT$.

Oricum, factorul de aproximare rezultă:

$$ALG \leq \max\{OPT, \left(\tfrac{3}{2} - \tfrac{1}{2m}\right) OPT\} = \left(\tfrac{3}{2} - \tfrac{1}{2m}\right) OPT$$

(deoarece $\frac{3}{2} - \frac{1}{2m} > 1$ pentru $m \geq 2$).

Ordered-Scheduling Algorithm este $\left(\frac{3}{2} - \frac{1}{2m}\right)$-aproximativ.

Problema 1 – TSP cu ponderi 1 și 2

Enunțul Problemei

Se consideră o variantă a problemei TSP pe un graf complet în care fiecare muchie are costul 1 sau 2. Se cer:

(a) Să se arate că TSP (pentru grafuri cu ponderi $\{1, 2\}$) este NP-hard.

(b) Să se demonstreze că valorile 1 și 2 satisfac inegalitatea triunghiului.

(c) Să se verifice dacă algoritmul de la curs (bazat pe calculul unui arbore de acoperire minimă și parcurgerea dublă a acestuia, cu shortcut-ing pentru a evita repetarea nodurilor) este $\frac{3}{2}$-aproximativ pentru această instanță a problemei TSP.

Rezolvare

(a) NP-hardness

Pentru a demonstra NP-hardness-ul, vom reduce problema ciclului hamiltonian (HC) – care este NP-completă – la TSP pentru grafuri cu ponderi 1 și 2.

Construcția:

- Fie $G = (V, E)$ un graf neponderat pentru care vrem să stabilim existența unui ciclu hamiltonian.
- Construiți un graf complet $G' = (V, E')$ astfel încât:
  - Dacă $\{u, v\} \in E$ (adică există o muchie între u și v în G), setați costul $c(u, v) = 1$ în G'.
  - Dacă $\{u, v\} \notin E$, setați $c(u, v) = 2$ în G'.

Argument:

- Dacă G are un ciclu hamiltonian, atunci în G' există un tur hamiltonian cu cost n (toate muchiile folosite au costul 1, unde $n = |V|$).
- Dacă G nu are ciclu hamiltonian, orice tur hamiltonian din G' va forța folosirea cel puțin a unei muchii de cost 2, astfel încât costul minim va fi cel puțin $n + 1$.

Prin urmare, dacă ar exista un algoritm polinomial care calculează soluția exactă pentru TSP pe grafuri cu costuri în $\{1, 2\}$, s-ar putea decide problema ciclului hamiltonian, ceea ce ar implica $P = NP$. Astfel, TSP cu ponderi $\{1, 2\}$ este NP-hard.

(b) Verificarea inegalității triunghiului

Trebuie să arătăm că pentru orice trei noduri $u, v, w \in V$ se are:

$$c(u, v) \leq c(u, w) + c(w, v).$$

Având în vedere că $c(u, v) \in \{1, 2\}$, analizăm următoarele cazuri:

Caz 1: $c(u, v) = 1$, $c(u, w) = 1$, $c(w, v) = 1$: $1 \leq 1 + 1 = 2$.

Caz 2: $c(u, v) = 1$, $c(u, w) = 1$, $c(w, v) = 2$: $1 \leq 1 + 2 = 3$.

Caz 3: $c(u, v) = 1$, $c(u, w) = 2$, $c(w, v) = 2$: $1 \leq 2 + 2 = 4$.

Caz 4: $c(u, v) = 2$: indiferent de valorile lui $c(u, w)$ și $c(w, v)$, cele mai mici valori posibile sunt 1 și 1, iar $1 + 1 = 2 \geq 2$.

În toate cazurile, inegalitatea triunghiului este satisfăcută.

(c) Aproximația algoritmului bazat pe MST

Algoritmul prezentat în curs pentru TSP (cu inegalitatea triunghiului) funcționează după următorii pași:

1. Se calculează un arbore de acoperire minimă (MST) T pe G'.
2. Se efectuează o parcurgere DFS a lui T pentru a construi un traseu Eulerian (fiecare muchie este parcursă de două ori), cu cost $\leq 2\,\text{cost}(T)$.
3. Se "scurtează" traseul (eliminând nodurile repetate) pentru a obține un tur hamiltonian H.

Analiză:

- Dacă G' are un ciclu hamiltonian cu cost n (adică, dacă toți pașii folosiți sunt cu cost 1), atunci $OPT = n$.
- În cel mai favorabil caz, MST-ul se formează folosind numai muchii de cost 1, deci $\text{cost}(T) \leq n - 1$. Parcurgerea DFS are cost $\leq 2(n - 1)$.
- După shortcuting (folosind inegalitatea triunghiului), costul final ALG este cel mult egal cu $2(n - 1)$.

Astfel, raportul de aproximare este:

$$\frac{ALG}{OPT} \leq \frac{2(n-1)}{n} = 2 - \frac{2}{n}.$$

Pentru valori mari ale lui n, raportul tinde spre 2, iar pentru valori mici (de exemplu, $n = 6$):

$$\frac{ALG}{OPT} \leq 2 - \frac{2}{6} = 2 - \frac{1}{3} \approx 1.67 > \frac{3}{2} = 1.5.$$

Concluzie: Chiar dacă în multe instanțe algoritmul (Christofides, care folosește și un matching pe nodurile de grad impar) poate atinge factorul $\frac{3}{2}$, algoritmul prezentat în curs (bazat pe MST și simpla parcurgere DFS) nu garantează factorul de aproximare $\frac{3}{2}$ pentru TSP pe grafuri cu ponderi $\{1, 2\}$. Există exemple concrete în care costul soluției obținute depășește $\frac{3}{2} \cdot OPT$.

Concluzie Generală

(a) TSP cu ponderi $\{1, 2\}$ este NP-hard (prin reducere de la HC).

(b) Ponderile 1 și 2 satisfac inegalitatea triunghiului.

(c) Algoritmul bazat pe MST (parcurgere DFS + shortcut) poate avea $\frac{ALG}{OPT} > \frac{3}{2}$, deci nu este $\frac{3}{2}$-aproximativ pe aceste instanțe.

Problema Vertex Cover în 3-CNF

Daniel Birsan

Enunțul Problemei

Fie $X = \{x_1, x_2, \ldots, x_n\}$ o mulțime de variabile booleene și fie Φ o formulă în forma normală conjunctivă (CNF) dată de

$$\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

unde fiecare clauză C_i este o disjuncție de exact 3 variabile (fără negare), adică

$$C_i = (x_{i_1} \vee x_{i_2} \vee x_{i_3}).$$

Observăm că dacă toate variabilele sunt setate ca true, formula este satisfăcută. Problema constă în a găsi un subset $S \subseteq X$ de variabile, cu cardinal minim, astfel încât, dacă atribuim valoarea true variabilelor din S (și false celorlalte), formula Φ este satisfăcută.

Se propune următorul algoritm (denumit Greedy-3CNF):

Pas 1: Fie $C = \{C_1, C_2, \ldots, C_m\}$ mulțimea de clauze, iar $X = \{x_1, x_2, \ldots, x_n\}$ mulțimea de variabile.

Pas 2: Cât timp $C \neq \emptyset$ execută:

(a) Alege aleator o clauză $C_j \in C$.

(b) Alege (în varianta de bază) o variabilă $x_i \in C_j$.

(c) Setează $x_i \leftarrow$ true.

(d) Elimină din C toate clauzele care conțin x_i.

Pas 3: Returnează mulțimea $S = \{x_i \in X \mid x_i = \text{true}\}$.

Observați că soluția optimă OPT este numărul minim de variabile care trebuie setate ca true pentru ca Φ să fie adevărată.

Rezolvarea

(a) Analiza factorului de aproximare pentru algoritmul de bază

În cel mai defavorabil caz, algoritmul de bază poate selecta, la fiecare pas, o variabilă care apare doar în clauza curentă și nu în altele.

Exemplu: Să considerăm clauzele:

$$C_1 = (x_1 \vee x_1 \vee x_2), \quad C_2 = (x_1 \vee x_1 \vee x_3), \quad \ldots, \quad C_{n-1} = (x_1 \vee x_1 \vee x_n).$$

Soluția optimă este să se seteze $x_1 = \text{true}$ (cost optim = 1), deoarece x_1 apare în toate clauzele. Totuși, algoritmul de bază, alegând aleator o variabilă din fiecare clauză, poate ajunge să aleagă $x_2, x_3, \ldots, x_n$ (fără a alege x_1), obținând astfel un cost de $n - 1$.

Concluzie (a): Factorul de aproximare al algoritmului de bază este de ordinul n (sau, mai exact, $n - 1$ în cel mai defavorabil caz).

(b) Modificarea algoritmului pentru a obține o aproximare de 3

Modificăm algoritmul astfel încât, la fiecare pas, pentru clauza aleasă $C_j = \{x_{i_1}, x_{i_2}, x_{i_3}\}$, setăm *toate* variabilele $x_{i_1}, x_{i_2}, x_{i_3}$ la true și eliminăm din C toate clauzele care conțin oricare dintre aceste variabile.

Observații: - Pentru orice clauză, soluția optimă trebuie să acopere clauza cu cel puțin o variabilă. - Dacă algoritmul nostru setează toate cele 3 variabile, în cel mai rău caz costul soluției poate fi de 3 ori costul optim (deoarece optim ar putea fi 1, iar algoritmul ar selecta 3).

Concluzie (b): Algoritmul modificat este 3-aproximativ.

(c) Reformularea problemei ca programare liniară

Problema Vertex Cover pentru 3-CNF se poate formula astfel:

Variabile: Fie $y_i \in [0, 1]$ asociată variabilei x_i, unde $y_i = 1$ înseamnă că x_i este setată la true.

Funcție de obiectiv:

$$\min \sum_{i=1}^{n} y_i.$$

Constrângeri: Pentru fiecare clauză $C_j = \{x_{i_1}, x_{i_2}, x_{i_3}\}$, impunem:

$$y_{i_1} + y_{i_2} + y_{i_3} \geq 1.$$

Aceasta asigură că cel puțin una din variabilele din fiecare clauză va fi setată la 1 (adevărat).

Concluzie (c): Problema se poate reformula ca următorul program liniar:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} y_i \\ \text{subject to} \quad & y_{i_1} + y_{i_2} + y_{i_3} \geq 1, \quad \forall\, C_j = \{x_{i_1}, x_{i_2}, x_{i_3}\}, \\ & 0 \leq y_i \leq 1, \quad i = 1, \ldots, n. \end{aligned}$$

(d) O soluție 3-aproximativă pentru problema de programare liniară

O metodă standard de rotunjire (rounding) este:

Pas 1: Se rezolvă programul liniar de mai sus pentru a obține soluția optimă fracțională y_i^* pentru $i = 1, \ldots, n$.

Pas 2: Se setează $x_i = \text{true}$ dacă $y_i^* \geq \frac{1}{3}$ și $x_i = \text{false}$ altfel.

Argumentare: Pentru orice clauză $C_j = \{x_{i_1}, x_{i_2}, x_{i_3}\}$, constrângerea are:

$$y_{i_1}^* + y_{i_2}^* + y_{i_3}^* \geq 1.$$

Astfel, cel puțin una dintre valorile $y_{i_k}^*$ trebuie să fie cel puțin $\frac{1}{3}$. Prin urmare, fiecare clauză este „acoperită" de cel puțin o variabilă rotunjită la true.

Costul soluției: Dacă $OPT_{LP} = \sum_{i=1}^{n} y_i^*$ este valoarea optimă a programului liniar, atunci mulțimea S obținută prin rotunjire are cost cel mult:

$$\sum_{i: y_i^* \geq \frac{1}{3}} 1 \leq 3 \cdot \sum_{i=1}^{n} y_i^* = 3 \cdot OPT_{LP}.$$

Deoarece $OPT_{LP} \leq OPT$ (soluția optimă integrala), soluția rotunjită este 3-aproximativă.

Concluzie (d): Rotunjind variabilele conform regulii $y_i^* \geq \frac{1}{3} \Rightarrow x_i = \text{true}$, se obține o soluție 3-aproximativă pentru problema Vertex Cover.

Rezumat Final

(a) Algoritmul de bază (alegere aleatorie a unei singure variabile) poate avea un factor de aproximare de ordin n (sau $n-1$) în cel mai defavorabil caz.

(b) Algoritmul modificat, în care se setează toate variabilele dintr-o clauză la true și se elimină toate clauzele ce conțin aceste variabile, este 3-aproximativ.

(c) Problema Vertex Cover în 3-CNF se poate reformula ca o problemă de programare liniară:

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} y_i \\ \text{s.t.} \quad & y_{i_1} + y_{i_2} + y_{i_3} \geq 1, \quad \forall\, C_j = \{x_{i_1}, x_{i_2}, x_{i_3}\}, \\ & 0 \leq y_i \leq 1, \quad \text{pentru } i = 1, \ldots, n. \end{aligned}$$

(d) Prin rotunjirea soluției fracționale: $x_i = \text{true}$ dacă $y_i^* \geq \frac{1}{3}$, se obține o soluție 3-aproximativă.

Algoritmul final pentru Vertex Cover este 3-aproximativ.