Proiect Analiza Algoritmilor: Problema Vertex Cover

Savu Vlad-Ștefan, Sandu Petru-Calin, Eduard Stefan Rusu

Facultatea de Automatică și Calculatoare,
Universitatea Politehnica din București

Rezumat Această lucrare analizează problema acoperirii cu vârfuri (Vertex Cover), o problemă fundamentală în teoria grafurilor și una dintre cele 21 de probleme NP-Complete identificate de Richard Karp în 1972. Studiul de față urmărește compararea performanței dintre o soluție exactă, bazată pe tehnica backtracking, și două abordări euristice/aproximative. Analiza se concentrează pe compromisul dintre timpul de execuție și calitatea soluției obținute (apropierea de cardinalitatea minimă).

Keywords: Vertex Cover · NP-Complete · Independent Set · Algoritmi Aproximativi · Backtracking · Optimizare combinatorie.

1 Introducere

1.1 Definirea Problemei

Problema acoperirii cu vârfuri (*Vertex Cover*) este o problemă clasică de optimizare a grafurilor. Fiind dat un graf neorientat $G = (V, E)$, o acoperire cu vârfuri este o submulțime de noduri $C \subseteq V$ astfel încât pentru fiecare muchie $\{u, v\} \in E$, cel puțin unul dintre capetele u sau v (sau ambele) aparține mulțimii C.

Scopul principal este identificarea unei acoperiri de cardinalitate minimă, numită *Minimum Vertex Cover*. În varianta sa de decizie, problema întreabă dacă există o acoperire de dimensiune cel mult k. Această variantă este demonstrată a fi NP-Complete, ceea ce implică faptul că, în absența unor dovezi contrare privind relația $P = NP$, nu există un algoritm de timp polinomial care să rezolve problema pe cazul general.

1.2 Context Istoric și Importanță

Vertex Cover ocupă un loc central în cercetarea complexității computaționale. Ea este strâns legată de alte probleme celebre, precum *Independent Set* și *Clique*. De fapt, o mulțime S este un set independent în graful G dacă și numai dacă complementul său, $V \setminus S$, este o acoperire cu vârfuri. Această dualitate este adesea utilizată în demonstrațiile de NP-Hard și în transformările polinomiale dintre probleme.

1.3 Aplicații practice și conexiuni cu alte probleme

Problema Vertex Cover nu este izolată; ea face parte dintr-o familie de probleme de optimizare combinatorie, servind drept model pentru numeroase situații reale:

- **Legătura cu Problema Rucsacului (Knapsack):** Deși Knapsack pare o problemă de inventar, în variantele sale pe grafuri, aceasta se intersectează cu Vertex Cover. Dacă fiecare nod are un cost de instalare diferit, problema devine una de tip Knapsack: selectarea subsetului de noduri cu cost minim care să acopere toate muchiile sub o constrângere de resurse.
- **Bioinformatică și Eliminarea Conflictelor:** În alinierea secvențelor de ADN, se construiesc grafuri unde muchiile reprezintă date contradictorii. Rezolvarea Vertex Cover permite identificarea setului minim de date ce trebuie eliminate pentru a obține un set consistent.
- **Cyber-Security (Sisteme IDS):** În rețelele mari, monitorizarea fiecărui pachet este costisitoare. Vertex Cover este utilizată pentru a plasa sisteme de detecție a intruziunilor în noduri strategice, asigurând monitorizarea tuturor conexiunilor.
- **Designul Circuitelor VLSI:** În proiectarea cipurilor, Vertex Cover ajută la minimizarea numărului de puncte de testare necesare pentru a verifica integritatea circuitelor pe suprafața siliciului.

2 Demonstrație NP-Complete

2.1 Apartenența la clasa NP

Pentru a demonstra că problema Vertex Cover aparține clasei NP, trebuie să arătăm că o **propunere de soluție** (un set de noduri) poate fi verificată în timp polinomial.

Presupunem că primim o submulțime de noduri $C \subseteq V$ despre care se afirmă că este o acoperire validă. Algoritmul de verificare parcurge fiecare muchie $\{u, v\} \in E$ și verifică dacă cel puțin unul dintre capete se află în setul C. Deoarece numărul de muchii este finit, iar verificarea fiecăreia se face rapid prin interogarea listei C, complexitatea totală este $O(E)$. Prin urmare, deoarece verificarea se face în timp polinomial, Vertex Cover aparține clasei NP.

2.2 Reducerea de la Independent Set

Demonstrăm că Vertex Cover este NP-Hard folosind o reducere polinomială de la problema *Independent Set*, despre care se știe deja că este NP-Hard.

Theorem 1. *Fie $G = (V, E)$ un graf neorientat. O mulțime $S \subseteq V$ este un set independent dacă și numai dacă complementul său, $V \setminus S$, este o acoperire cu vârfuri (Vertex Cover) pentru graful G.*

Demonstrație. ($\Rightarrow$) Presupunem că S este un set independent. Conform definiției, nicio muchie din E nu are ambele capete în S. Acest lucru înseamnă că pentru orice muchie $\{u, v\}$, cel puțin unul dintre noduri (u sau v) trebuie să se afle în afara lui S, adică în $V \setminus S$. Astfel, $V \setminus S$ acoperă toate muchiile, deci este un Vertex Cover.

($\Leftarrow$) Presupunem că $V \setminus S$ este o acoperire cu vârfuri. Atunci, orice muchie din graf are cel puțin un capăt în $V \setminus S$. Rezultă că nicio muchie nu poate avea ambele capete în S. Aceasta este exact definiția unui set independent, deci S este un set independent.

Reducere (varianta de decizie). Consideram instanta INDEPENDENT SET (G, k). Construim in timp polinomial instanta VERTEX COVER (G, k'), unde $k' = |V| - k$ (pasul de transformare este doar calculul lui k' si pastrarea aceluiasi graf). Din Teorema 1 rezulta echivalenta:

G are un independent set de marime $\geq k \iff G$ are un vertex cover de marime $\leq |V| - k = k'$.

Prin urmare, deoarece transformarea este polinomiala si INDEPENDENT SET este NP-Complete, rezulta ca VERTEX COVER este NP-Hard.

3 Algoritmi pentru problema Vertex Cover

Având în vedere caracterul NP-Complete al problemei Vertex Cover, abordările algoritmice se împart în două mari categorii: algoritmi exacți, care garantează obținerea soluției optime dar au complexitate exponențială, și algoritmi aproximativi sau euristici, care sacrifică exactitatea în favoarea unui timp de execuție redus. În această secțiune sunt analizate un algoritm exact bazat pe backtracking (Branch & Bound) și două abordări aproximative clasice.

3.1 Algoritm exact bazat pe Backtracking (Branch & Bound)

Ideea de bază Algoritmul exact pornește de la observația fundamentală conform căreia, pentru orice muchie $\{u, v\} \in E$, cel puțin unul dintre noduri trebuie să aparțină acoperirii. Această constrângere permite construirea unui arbore de decizie binar, în care fiecare ramură corespunde includerii unuia dintre capetele muchiei selectate.

Tehnica Branch & Bound extinde backtracking-ul clasic prin introducerea unor limite (bound-uri) care permit eliminarea timpurie a ramurilor ce nu pot conduce la o soluție mai bună decât cea deja cunoscută.

Descrierea algoritmului Algoritmul funcționează recursiv după următorii pași:

1. Se selectează o muchie neacoperită $\{u, v\}$.
2. Se creează două ramuri:

   - se include u în soluție;
   - se include v în soluție.
3. Se elimină din graf muchiile acoperite de nodul ales.
4. Se continuă recursiv până când nu mai există muchii neacoperite.
5. Ramurile pentru care cardinalitatea soluției parțiale depășește cea mai bună soluție cunoscută sunt eliminate (pruning).

```
BranchAndBound(G, C):
    if E(G) = ∅:
        actualizeaza solutia optima
        return

    if |C| ≥ best:
        return

    alege muchia (u, v)

    BranchAndBound(G - u, C ∪ {u})
    BranchAndBound(G - v, C ∪ {v})
```

Complexitate

- **Timp:** $O(2^{|V|})$ în cel mai defavorabil caz.
- **Spațiu:** $O(|V|)$ datorită adâncimii maxime a recursiei.

Acest algoritm este aplicabil exclusiv grafurilor de dimensiuni mici, fiind utilizat în special în contexte educaționale sau pentru validarea soluțiilor aproximative.

Avantaje și dezavantaje Avantaje:

- Produce întotdeauna o soluție optimă pentru problema Vertex Cover.
- Permite eliminarea eficientă a ramurilor nepromițătoare prin tehnica de pruning (Branch & Bound).
- Este util pentru validarea soluțiilor aproximative sau pentru grafuri foarte mici.

Dezavantaje:

- Are complexitate exponențială în cel mai defavorabil caz.
- Nu este scalabil pentru grafuri de dimensiuni medii sau mari.
- Consumul de timp crește rapid odată cu numărul de noduri.

3.2 Algoritm aproximativ 2-approx bazat pe muchii

Principiul algoritmului Algoritmul aproximativ clasic pentru Vertex Cover se bazează pe selectarea repetată a unei muchii arbitrare și includerea ambelor capete ale acesteia în acoperire. Muchiile incidente acestor noduri sunt apoi eliminate din graf.

Această strategie este echivalentă cu construirea unei potriviri maxime (maximal matching), iar acoperirea rezultată este formată din toate nodurile potrivirii.

```
ApproxVertexCover(G):
    C <- empty
    while E(G) ≠ ∅:
        alege muchia (u, v)
        C <- C ∪ {u, v}
        elimina toate muchiile incidente lui u sau v
    return C
```

Garanția de aproximare Fie OPT cardinalitatea unei acoperiri minime și ALG soluția produsă de algoritm. Atunci:

$$ALG \leq 2 \cdot OPT$$

Această limită este strict demonstrabilă, deoarece fiecare muchie selectată trebuie să fie acoperită de cel puțin un nod în soluția optimă, iar algoritmul include cel mult două.

Complexitate

- **Timp:** $O(|E|)$
- **Spațiu:** $O(|V|)$

Avantaje și dezavantaje Avantaje:

- Rulează în timp polinomial, fiind eficient chiar și pentru grafuri mari.
- Are o garanție teoretică de aproximare de 2.
- Este simplu de implementat și de explicat.

Dezavantaje:

- Nu garantează obținerea unei soluții optime.
- Calitatea soluției poate fi semnificativ mai slabă decât optimul în unele cazuri.
- Alegerea arbitrară a muchiilor poate influența rezultatul final.

3.3 Algoritm exact parametrizat (FPT): Buss kernel + bounded search

Ideea de baza Folosim o abordare FPT pentru varianta de decizie a problemei Vertex Cover: dat un graf $G = (V, E)$ si un parametru k, vrem sa decidem daca exista un vertex cover de dimensiune cel mult k. Algoritmul combina doua componente:

- **Kernelizare (Buss)**: aplicam repetat reguli de reducere care simplifica graful folosind parametrul k;
- **Bounded search tree**: daca dupa reduceri mai raman muchii, ramificam pe o muchie (u, v) si incercam cele doua optiuni posibile (trebuie sa alegem cel putin unul dintre capetele muchiei).

Pentru a obtine **solutia minima**, nu presupunem k cunoscut: rulam procedura de decizie pentru valori crescatoare ale lui k (de la un lower bound pana la un upper bound) si prima valoare k pentru care raspunsul este *DA* produce o solutie optima.

Reguli de reducere (kernelizare Buss) Aplicam iterativ urmatoarele reguli pana la stabilizare:

1. **Grad mare:** daca exista un varf v cu $\deg(v) > k$, atunci v trebuie inclus in orice vertex cover de dimensiune $\leq k$. Il adaugam in solutie, eliminam toate muchiile incidente lui v si decrementam k.
2. **Grad 1:** daca exista un varf v cu $\deg(v) = 1$ si vecin unic u, includem u in solutie, eliminam muchiile incidente lui u si decrementam k.
3. **Buss bound pe muchii:** dupa aplicarea regulii de grad mare, in graful ramas toate varfurile au grad $\leq k$. In acest caz, un cover de dimensiune $\leq k$ poate acoperi cel mult $k \cdot k$ muchii, deci daca $|E| > k^2$ raspunsul este *NU*.

Cautare recursiva (bounded search tree) Daca dupa reduceri graful mai contine muchii si $k > 0$, alegem o muchie $(u, v) \in E$. Orice vertex cover trebuie sa contina cel putin unul dintre capetele muchiei, deci ramificam:

- includem u si apelam recursiv pe graful fara muchiile incidente lui u cu $k \leftarrow k - 1$;
- includem v si apelam recursiv pe graful fara muchiile incidente lui v cu $k \leftarrow k - 1$.

```
DECIDE_VC(G, k):
    // deg(v)>k => fortat; deg(v)=1 => ia vecinul; daca |E|>k^2 => NO
    aplica_reduceri_Buss(G, k)

    if k < 0: return NO
    if E(G) este vida: return YES
```

```
    alege o muchie (u,v) din E(G)
    return DECIDE_VC(G - inc(u), k-1) OR DECIDE_VC(G - inc(v), k-1)

MIN_VC(G):
    LB = lower_bound_matching(G)
    UB = upper_bound_2approx(G)
    for k = LB .. UB:
        if DECIDE_VC(G, k) == YES:
            return k
```

Complexitate

- **Timp:** in cel mai rau caz, ramificarea produce un arbore de cautare de dimensiune $O(2^k)$, iar fiecare pas contine operatii polinomiale (reduceri + actualizari). Prin urmare, complexitatea este de forma $O(2^k \cdot \text{poly}(|V|, |E|))$.
- **Spatiu:** $O(k + |V| + |E|)$ pentru starea curenta si solutia (kernelul ramas are dimensiune controlata de k).

Avantaje / Dezavantaje

- **Avantaje:**
  - Permite tratarea grafurilor mari atunci când soluția optimă este mică.
  - Combină eficiența reducerii cu exactitatea algoritmului Branch & Bound.
- **Dezavantaje:**
  - Necesită cunoștințe avansate de algoritmi parametrizați.
  - Ineficient dacă parametrul k nu este mic.

3.4 Relaxare liniară și rotunjire

Formulare prin programare liniară Problema Vertex Cover poate fi formulată ca un program liniar întreg, în care fiecărui nod $v \in V$ îi este asociată o variabilă binară x_v, ce indică dacă nodul este inclus sau nu în acoperire:

$$x_v = \begin{cases} 1, & \text{dacă } v \in C \\ 0, & \text{altfel} \end{cases}$$

Funcția obiectiv urmărește minimizarea cardinalității acoperirii:

$$\min \sum_{v \in V} x_v$$

Constrângerile asigură faptul că fiecare muchie este acoperită:

$$x_u + x_v \geq 1 \quad \forall \{u, v\} \in E$$

$$x_v \in \{0, 1\}$$

Această formulare este echivalentă cu problema originală, însă fiind un program liniar întreg, rămâne NP-Hard.

Prin relaxarea condiției de integritate și permiterea valorilor fracționare:

$$x_v \in [0, 1]$$

problema devine un program liniar clasic, care poate fi rezolvat în timp polinomial folosind metode precum algoritmul simplex sau metode de punct interior.

Rotunjirea soluției Soluția fracționară obținută prin relaxare liniară nu reprezintă, în general, o acoperire validă, deoarece valorile variabilelor nu mai sunt binare. Pentru a obține o soluție fezabilă pentru problema Vertex Cover, se aplică o tehnică de rotunjire.

Regula standard de rotunjire este:

$$x_v \geq 0.5 \Rightarrow v \in C$$

Această regulă garantează că pentru orice muchie $\{u, v\}$, cel puțin unul dintre capete va fi inclus în acoperire, deoarece constrângerea $x_u + x_v \geq 1$ implică faptul că cel puțin una dintre variabile este mai mare sau egală cu 0.5.

Complexitate

- **Timp:** $O(\text{LP}(V, E))$, unde $\text{LP}(V, E)$ reprezintă timpul de rezolvare al programului liniar, polinomial în $|V|$ și $|E|$.
- **Spațiu:** $O(|V| + |E|)$.

Avantaje și dezavantaje Avantaje:

- Problema relaxată poate fi rezolvată în timp polinomial folosind algoritmi standard de programare liniară.
- Oferă o garanție teoretică de aproximare de 2 pentru problema Vertex Cover.
- Abordarea este bazată pe fundamente matematice solide și permite extinderi către alte probleme de optimizare combinatorială.
- Soluția fracționară poate oferi informații utile despre structura soluției optime.

Dezavantaje:

- Necesită utilizarea unui solver de programare liniară, ceea ce poate crește complexitatea implementării.
- Soluția obținută după rotunjire nu este optimă în general.
- Pentru grafuri foarte mari, costul rezolvării programului liniar poate deveni semnificativ.

4 Evaluare experimentala

4.1 Conditii de testare

Testele au fost rulate pe urmatoarea configuratie:

- **Hardware:** `Ryzen 7 8700G`, `8N/16T`, `64GB ram`.
- **Sistem de operare:** `<Nobara Linux 43> <Linux 6.18.2-200.nobara.fc43.4>`.
- **Compilator:** C++: `-O2 -std=c++17 -Wall -Wextra -pedantic -DUSE_GLPK`.
- **Solver LP:** GLPK (compilare cu `-DUSE_GLPK` si link `-lglpk`).
- **Timeout:** `4000` ms pentru algoritmii exacati (BB si FPT).
- **Repetari:** `5` repetari/instanta; timpul raportat este `median`.
- **Post-procesare solutie:** `cleanup_cover`.
- **Set de teste:** `tests/`, total `40` instante.

4.2 Test suite si generare date

Pentru evaluarea algoritmilor am folosit o suita determinista de teste, generata automat cu un script Python. Scopul a fost sa acoperim atat cazuri mici (unde putem valida usor corectitudinea), cat si instante mai mari (unde putem observa scalarea timpului de executie). Testele sunt scrise in fisiere text separate, numerotate, pentru a putea reproduce rezultatele in mod consistent.

Formatul fisierelor de intrare Fiecare test este un fisier `XX.in` care descrie un graf neorientat:

- Prima linie contine doua numere intregi: `N M`, unde `N` este numarul de noduri, iar `M` este numarul de muchii.
- Urmeaza `M` linii de forma `u v`, cu `u` si `v` id-uri 0-based pentru capetele muchiei.

Compozitia suitei de teste (grupare pe intervale 01–40) Suita contine 40 de instante, gandite sa acopere cazuri de baza, topologii clasice, grafuri regulate, stres pe dimensiune si cateva teste speciale unde ordinea muchiilor poate influenta implementari greedy (de tip maximal matching).

- **Teste 01–04 (cazuri foarte mici / sanity checks):** grafuri triviale si verificabile manual: fara muchii, o muchie, lant scurt, ciclu mic.
- **Teste 05–09 (structuri canonice mici-medii):** clica K_5, stea mica, lant pe 20 noduri, ciclu pe 20 noduri, bipartit complet $K_{10,10}$.
- **Teste 10–12 (densitate controlata + componente multiple):** lant pe 100 noduri cu cateva muchii suplimentare (chords), graf relativ dens pe 100 noduri (numar fix de muchii), si un test cu 3 componente conexe (lant + ciclu + stea) in acelasi fisier.
- **Teste 13–16 (structuri regulate / sparse control):** arbore (layout de arbore binar), grid 10×10, clica K_{15}, si un graf bipartit rar pe 200 noduri (muchii distribuite determinist).

- **Teste 17–21 (stres pe dimensiune si topologii extreme):** graf cu hub-uri (3 hub-uri conectate la majoritatea nodurilor), lant cu 1000 noduri, stea cu 1000 noduri, graf pe 500 noduri cu un numar fix de muchii (chunk deterministic), grid subtire 2×250.
- **Teste 22–25 (dense + edge cases + structuri compuse):** doua clici mari legate printr-un pod, graf gol (fara muchii), graf pe 250 noduri cu un numar fix de muchii, si o padure (4 lanturi) cu cateva legaturi intre componente.
- **Teste 26–27 (windmill / friendship graph, sensibil la ordinea muchiilor):** aceeasi instanta F_k scrisa in doua ordine: una "rea" (muchiile frunza-frunza primele) si una "buna" (muchiile cu hub-ul primele). Scopul este sa evidentieze dependenta de ordine pentru unele euristici greedy.
- **Teste 28–30 (grafuri mici dar structurale, pentru comportament non-trivial):** graf circulant $C(n; 1, 2, 7)$ (multe cicluri/triunghiuri), lollipop (clica K_{20} + coada de lant), si un graf "aproape bipartit" ($K_{12,12}$ plus un ciclu impar in partea stanga).
- **Teste 31–32 (Erdos-Renyi determinist, densitati diferite):** grafuri $G(n, p)$ pe 120 noduri cu seed fix, pentru doua valori ale lui p (mai rar vs. mai dens), utile pentru compararea timpului si a raportului fata de optim pe instante "random-like".
- **Test 33 (cover plantat, OPT cunoscut):** instanta construita astfel incat exista un vertex cover de dimensiune exacta k (muchiile sunt construite sa atinga un set C de dimensiune k, iar intre nodurile din afara lui C nu exista muchii). Util pentru validarea corectitudinii si pentru verificarea calitatii aproximarii.
- **Test 34 (barbell):** doua clici mari conectate printr-un lant (nu doar un pod). Este un exemplu clasic cu subgraf dens + legatura rara, util pentru stres pe algoritmi exacti.
- **Test 35 (3-partite dense-ish):** graf cu trei partitii (A,B,C) si multe muchii intre ele, cu rarire determinista. Produce multe triunghiuri, dar nu este clica; bun pentru a evita "trivialitatea" bipartita.
- **Test 36 (bipartit cu ordine de muchii nefavorabila):** graf bipartit stratificat, apoi muchiile sunt reordonate (grupate dupa partea dreapta) pentru a crea un parcurs nefavorabil pentru selectia greedy.
- **Teste 37–38 (acelasi graf, ordine inversata):** aceeasi instanta $G(n, p)$ pe 160 noduri, salvata in ordine lexicografica vs. ordine inversa. Permite izolarea efectului ordinii de intrare asupra euristicilor.
- **Teste 39–40 (acelasi graf structurat, ordine buna vs. shuffle):** graf cu un nucleu dens (clica) + coada rara (lant) + legaturi catre cateva noduri hub; aceeasi multime de muchii este scrisa o data in ordine "buna" (determinista) si o data amestecata cu un seed fix.

Ordinea muchiilor ca factor experimental Desi problema Vertex Cover este definita pe graf, unele implementari greedy (ex.: selectarea unei muchii arbitrare sau maximal matching) pot fi influentate de ordinea in care sunt citite

muchiile. De aceea am inclus teste pereche cu **aceeasi multime de muchii**, dar cu **ordine diferita**:

- **Windmill / friendship graph** F_k: o ordine "rea" (muchiile intre frunze primele) vs. o ordine "buna" (muchiile cu hub-ul primele).
- **Acelasi graf, ordine inversata:** aceeasi instanta scrisa lexicografic vs. invers.
- **Ordine amestecata determinist:** aceeasi instanta cu muchii amestecate cu un seed fix.

Reproducibilitate Scriptul de generare este determinist (foloseste reguli fixe si seed-uri explicite acolo unde este cazul), astfel incat suita poate fi regenerata identic pe orice sistem. Aceasta proprietate ajuta la compararea corecta a algoritmilor si la depanare.

4.3 Date Experimentale:

Tabela 1: Timpi de executie (ms). `TO` indica depasirea timeout-ului. Valorile sunt mediane pe `5` rulari.

Test	n	m	BB (ms)	FPT (ms)	MATCH (ms)	LP (ms)
01.in	1	0	0.000900	**0.000000**	0.000050	0.000020
02.in	2	1	0.000560	**0.000000**	0.000110	0.104878
03.in	3	2	0.000360	**0.000000**	0.000070	0.036509
04.in	4	4	0.000610	0.000410	**0.000090**	0.035879
05.in	5	10	0.001720	0.001820	**0.000110**	0.042329
06.in	10	9	0.000460	0.000210	**0.000070**	0.033049
07.in	20	19	0.001480	0.000660	**0.000250**	0.045329
08.in	20	20	0.000760	0.000540	**0.000240**	0.052289
09.in	20	100	0.001570	0.001090	**0.000290**	0.226756
10.in	100	120	0.052479	0.905691	**0.001370**	0.336153
11.in	100	2000	1711.191803	1.418921	**0.000920**	8.422586
12.in	20	18	0.001970	0.002960	**0.000190**	0.051529
13.in	50	49	0.007030	0.008050	**0.000440**	0.097988
14.in	100	180	0.003799	0.002490	**0.001290**	0.522090
15.in	15	105	0.482130	0.010340	**0.000240**	0.133227
16.in	200	100	0.005400	0.003870	**0.002460**	0.360132
17.in	100	294	0.001310	0.000830	**0.000160**	0.151237
18.in	1000	999	0.028859	0.024350	**0.017390**	10.216730
19.in	1000	999	0.003080	0.002180	**0.000480**	0.698336
20.in	500	2500	0.037539	0.841272	**0.000550**	2.051598
21.in	500	748	0.014940	0.012380	**0.008149**	5.886419

Continua pe pagina urmatoare

Test	n	m	BB (ms)	FPT (ms)	MATCH (ms)	LP (ms)
22.in	200	9901	TO	TO	**0.034190**	51.405499
23.in	30	0	0.000500	0.000300	0.000090	**0.000060**
24.in	250	1000	0.011400	0.112918	**0.000250**	0.562889
25.in	200	198	0.005980	0.004340	**0.002519**	0.651817
26.in	121	180	0.020830	0.103518	**0.001490**	0.584118
27.in	121	180	0.003970	0.103208	**0.001560**	0.545319
28.in	30	90	0.212586	11.646000	**0.000780**	0.250045
29.in	40	210	25.156662	0.102158	**0.000640**	0.427821
30.in	24	149	0.005240	0.008200	**0.000370**	0.343832
31.in	120	597	TO	TO	**0.002410**	2.839111
32.in	120	1294	TO	TO	**0.003630**	5.414659
33.in	200	548	TO	1.660176	**0.000810**	0.769084
34.in	100	911	TO	TO	**0.005270**	2.771063
35.in	120	3986	TO	TO	**0.003940**	87.914548
36.in	120	240	0.004440	0.003130	**0.001720**	0.961600
37.in	160	1335	TO	TO	**0.004190**	8.165512
38.in	160	1335	TO	TO	**0.004339**	6.033676
39.in	200	1047	TO	900.190453	**0.004590**	3.990588
40.in	200	1047	TO	TO	**0.004130**	4.566826

Tabela 2: Calitatea solutiilor pentru MATCH si LP. Raportul este calculat fata de OPT atunci cand OPT este cunoscut; altfel raportul este `NA`. Raportam `raw` si dupa `cleanup`.

Test	OPT	MATCH raw	MATCH	Ratio	LP raw	LP	Ratio
01.in	0	0	**0**	NA	0	**0**	NA
02.in	1	2	**1**	**1.000000**	1	**1**	**1.000000**
03.in	1	2	**1**	**1.000000**	1	**1**	**1.000000**
04.in	2	4	**2**	**1.000000**	2	**2**	**1.000000**
05.in	4	4	**4**	**1.000000**	5	**4**	**1.000000**
06.in	1	2	**1**	**1.000000**	1	**1**	**1.000000**
07.in	10	20	**10**	**1.000000**	10	**10**	**1.000000**
08.in	10	20	**10**	**1.000000**	10	**10**	**1.000000**
09.in	10	20	**10**	**1.000000**	10	**10**	**1.000000**
10.in	53	100	**53**	**1.000000**	61	**53**	**1.000000**
11.in	23	24	**23**	**1.000000**	23	**23**	**1.000000**
12.in	7	12	**7**	**1.000000**	10	**7**	**1.000000**
13.in	18	34	20	1.111111	18	**18**	**1.000000**
14.in	50	100	**50**	**1.000000**	50	**50**	**1.000000**

Continua pe pagina urmatoare

Test	OPT	MATCH raw	MATCH	Ratio	LP raw	LP	Ratio
15.in	14	14	**14**	**1.000000**	15	**14**	**1.000000**
16.in	100	200	**100**	**1.000000**	100	**100**	**1.000000**
17.in	3	6	**3**	**1.000000**	3	**3**	**1.000000**
18.in	500	1000	**500**	**1.000000**	500	**500**	**1.000000**
19.in	1	2	**1**	**1.000000**	1	**1**	**1.000000**
20.in	6	6	**6**	**1.000000**	6	**6**	**1.000000**
21.in	250	500	**250**	**1.000000**	250	**250**	**1.000000**
22.in	198	200	**198**	**1.000000**	200	**198**	**1.000000**
23.in	0	0	**0**	**1.000000**	0	**0**	**1.000000**
24.in	5	6	**5**	**1.000000**	5	**5**	**1.000000**
25.in	100	200	**100**	**1.000000**	100	**100**	**1.000000**
26.in	61	120	**120**	**1.967213**	121	**120**	**1.967213**
27.in	61	120	**61**	**1.000000**	121	120	1.967213
28.in	20	30	**20**	**1.000000**	30	**20**	**1.000000**
29.in	29	40	**29**	**1.000000**	39	**29**	**1.000000**
30.in	12	20	**12**	**1.000000**	12	**12**	**1.000000**
31.in	NA	110	**90**	NA	120	92	NA
32.in	NA	118	**104**	NA	120	**104**	NA
33.in	30	32	**30**	**1.000000**	30	**30**	**1.000000**
34.in	NA	100	**78**	NA	100	**78**	NA
35.in	NA	80	**80**	NA	120	**80**	NA
36.in	60	120	**60**	**1.000000**	60	**60**	**1.000000**
37.in	NA	150	136	NA	160	**131**	NA
38.in	NA	156	133	NA	160	**131**	NA
39.in	119	200	**146**	**1.226891**	200	**146**	**1.226891**
40.in	NA	176	**130**	NA	200	146	NA

5 Concluzii

In aceasta sectiune discutam datele obtinute (Tabelul 1 si Tabelul 2), le comparam cu asteptarile teoretice din sectiunile anterioare si formulam recomandari pentru utilizare in situatii reale. Am folosit patru abordari: un algoritm exact de tip Branch & Bound (BB), o varianta exacta parametrizata (FPT / kernelization + rezolvare exacta pe kernel), o euristica 2-approx bazata pe matching (MATCH) si relaxarea liniara cu rotunjire (LP).

Observatii privind timpii de executie (scalare si praguri) (1) BB: comportament exponential, dar rapid pe grafuri mici/structurate. Conform analizei de complexitate, BB are comportament exponential in cel mai defavorabil caz, iar rezultatele confirma acest lucru: pentru grafuri mici (ex. teste 01–21,

24–30, 36) timpii sunt foarte mici, dar pe grafuri dense sau combinatorial dificile apar depasiri de timeout (`TO`). Se observa doua tipare:

- **Densitatea mare produce explozie combinatorica:** de exemplu, un graf foarte dens (testul 22 cu doua clici mari conectate) duce la `TO` pentru BB.
- **Grafuri aleatoare/mediu-dense sunt adesea cele mai grele:** pe instante de tip $G(n, p)$ (teste 31, 32, 34, 35, 37, 38, 40) BB intra in `TO`, ceea ce este in linie cu faptul ca astfel de grafuri nu au structura usoara (arbore, lant, stea) care sa ajute pruning-ul.

Acest rezultat confirma asteptarea: BB este excelent ca "ground truth" pe instante mici sau pe grafuri cu structura simpla, dar nu este robust ca metoda generala pentru grafuri mari si/sau dense.

(2) FPT / kernelization: robust pe multe instante, dar poate deveni lent cand parametrul efectiv nu e mic. Teoretic, kernelization ofera castiguri cand k (dimensiunea solutiei) este relativ mic raportat la n, deoarece reduce problema la un kernel de marime $O(k^2)$. In datele experimentale, FPT rezolva foarte rapid multe teste si ramane fezabil chiar si acolo unde BB da `TO` (ex. testul 33, unde $OPT = 30$ e mic raportat la $n = 200$). Totusi, apar si cazuri in care FPT depaseste timeout (ex. teste 31, 32, 34, 35, 37, 38, 40), ceea ce sugereaza ca in acele instante:

- fie k este mare (sau efectiv mare) si kernelul rezultat nu mai este mic,
- fie regulile de reducere nu comprima suficient graful, iar pasul exact ramane greu.

Asta este in linie cu teoria: FPT nu promite eficienta pentru valori mari ale parametrului, ci pentru cazurile unde exista o solutie relativ mica.

(3) MATCH si LP: timpi foarte mici si scalare buna. Atat euristica bazata pe matching (MATCH) cat si relaxarea liniara + rotunjire (LP) sunt polinomiale; timpii obtinuti sunt foarte mici comparativ cu metodele exacte si raman stabili pe instante mari (ex. teste 18–21 si chiar pe instante dense). In mod practic, acestea sunt abordari predictibile ca runtime.

Calitatea solutiilor: potrivire cu garantiile (1) Cazuri unde OPT este cunoscut: MATCH si LP respecta comportamentul asteptat. Din Tabelul 2, pe majoritatea testelor unde OPT este cunoscut, raportul final (`cleanup`) este 1.0, adica solutia coincide cu optimul. Acest lucru nu contrazice teoria (garantia de 2-approx este o limita superioara, nu o medie): pe multe grafuri standard (lanturi, cicluri, stele, bipartite complete, grile), aproximarile pot fi optime.

(2) Efectul ordinii muchiilor (teste 26–27) confirma un risc practic pentru implementari greedy. Teste 26 si 27 sunt construite astfel incat multimea de muchii este aceeasi, dar ordinea difera. Rezultatul confirma asteptarea:

- in testul 26 (ordine nefavorabila), MATCH ajunge la un rezultat mult mai slab dupa cleanup (raport ≈ 1.97 fata de OPT),

– in testul 27 (ordine favorabila), MATCH obtine solutia optima (raport 1.0).

Aici se vede clar ca implementarea greedy a unui matching maximal poate fi sensibila la ordinea de parcurgere a muchiilor, deci in situatii reale (date venite din fisiere/loguri/streaming) rezultatul poate varia. Acest lucru merita mentionat explicit ca limitare practica pentru MATCH daca nu se aplica o randomizare controlata sau o strategie de alegere mai robusta.

(3) LP poate fi mai stabil ca output, dar nu este imun la decalaje. In general, LP dupa rotunjire are tot garantie de 2-approx. In testele unde OPT este cunoscut, LP a fost de cele mai multe ori optim, dar exista cazuri unde solutia LP este mai mare decat OPT la nivel `raw` (de exemplu in grafuri clique-like), iar dupa `cleanup` poate ramane la un raport ¿ 1. In plus, LP introduce dependenta de solver (si costuri de implementare), deci trade-off-ul este: *solutii adesea bune si mai stabile, dar cu infrastructura suplimentara.*

Comparatie directa cu asteptarile din sectiunile anterioare Rezultatele sunt consistente cu discutia teoretica:

– **Algoritmii exacti** (BB, respectiv FPT) sunt corecti, dar runtime-ul explodeaza pe instante dense sau fara structura favorabila; FPT ajuta mai ales cand k este mic.
– **Algoritmii aproximativi** (MATCH si LP) au runtime predictibil si foarte mic; calitatea solutiilor este de multe ori excelenta pe grafuri structurale, iar limita teoretica (factor 2) este atinsa doar in cazuri construite adversarial sau nefavorabile (precum testele cu ordine de muchii proasta).

Recomandari de utilizare in cazuri reale In aplicatii reale, alegerea metodei depinde de marimea grafului, densitate si de cat de important este optimul exact.

Cand folosim BB (exact):

– grafuri mici (zeci de noduri) sau cand trebuie validata o solutie (benchmark/ground truth),
– cand exista structura puternica (de exemplu grafuri de tip arbore + putine muchii suplimentare) si pruning-ul functioneaza bine.

Cand folosim FPT / kernelization (exact parametrizat):

– cand se suspecteaza ca solutia optima este relativ mica (de exemplu, “putine noduri-cheie” care acopera majoritatea conexiunilor),
– in scenarii de tip “curatare de conflicte” unde dorim sa eliminam un set mic de elemente pentru a elimina toate conflictele.

Cand folosim MATCH (2-approx, foarte rapid):

– pentru grafuri foarte mari unde avem nevoie de o solutie imediata,
– pentru pipeline-uri repetitive (multe instante), unde costul per instanta trebuie sa fie foarte mic,

– **atentie:** daca input-ul are ordine arbitrara a muchiilor, rezultatul poate varia; e recomandata o reordonare determinista (ex. sortare) sau randomizare cu seed fix pentru stabilitate.

Cand folosim LP (2-approx, stabil, dar cu solver):

– cand vrem o solutie de calitate buna, relativ stabila, si avem acces la un solver LP,
– cand vrem sa folosim informatia din solutia fractionara (ex. valori aproape de 0.5) ca semnal pentru post-procesari sau euristici hibride.

Concluzie scurta Experimental, metodele aproximative sunt cele mai practice ca runtime, iar metodele exacte sunt utile ca referinta sau cand instanta este mica / are parametrul mic. Un aspect important validat de teste este sensibilitatea euristicii MATCH la ordinea muchiilor, ceea ce trebuie controlat explicit in implementari reale pentru rezultate reproductibile.

Bibliografie

1. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972). *(Sursa principală pentru clasificarea Vertex Cover ca fiind NP-Complete.)*
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009). *(Manualul standard de algoritmică folosit pentru definițiile formale și demonstrația reducerii de la Independent Set.)*
3. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco (1979). *(Cartea de referință pentru demonstrarea NP-Hard diverselor probleme.)*
4. Sipser, M.: Introduction to the Theory of Computation, 3rd edn. Cengage Learning (2012). *(Sursa pentru conceptele de clase de complexitate, certificate și verificatori.)*
5. GeeksforGeeks: Vertex Cover Problem — Set 1 (Introduction and Approximate Algorithm), `https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm/`. Accesat la: 2024-05-22. *(Sursa pentru implementarea euristică de bază.)*
6. V. Vazirani, *Approximation Algorithms*, Springer, 2001. *(Sursa pentru demonstrația teoretică a algoritmului 2-approx și pentru relaxarea liniară și rotunjire.)*
7. R. Niedermeier, *Invitation to Fixed-Parameter Algorithms*, Oxford University Press, 2006. *(Sursa pentru kernelization în Vertex Cover și reducerea la kernel de dimensiune $O(k^2)$.)*