Reducere kClique $\leq_P$ SAT

Tăiatu Iulian - 322CB

1 Introducere

1.1 Notiuni generale

In cele ce urmeaza, notatia $G = (V, E)$ va fi intepretata astfel: *Graful G este format din nodurile V si muchiile E*, unde $V = \{v_1, v_2, \ldots, v_n\}$, iar $E = \{e_1, e_2, \ldots, e_m\}$.

Ordinul unui graf este data de numarul de noduri ($|V|$), iar *dimensiunea* este data de numarul de muchii ($|E|$).

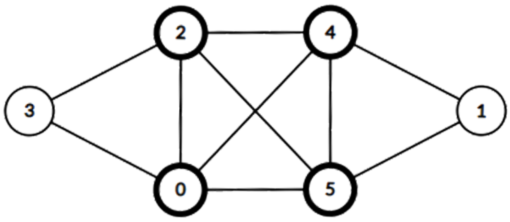
Spunem ca un graf este **complet** daca fiecare pereche de noduri distincte este conectata printr-o muchie. Putem formaliza astfel: $\forall\ u, v \in V\ (u \neq v),\ (u, v)\ ||\ (v, u) \in E$. *Observatie:* Consideram ambele perechi (u, v), respectiv (v, u), deoarece discutia va avea loc in contextul *grafurilor neorientate*.

Observatie: Pe parcursul lucrarii, vor exista link-uri catre materiale aditionale, pentru o aprofundare mai buna a subiectului.

1.2 Problema kClique

Problema **kClique**[1] primeste ca input un graf, un numar intreg k si intoarce true daca exista un *subgraf complet de dimensiune k* si false in caz contrar.

Exemplu: Graful din figura contine o clica de dimensiune 4, formata din nodurile $\{0, 2, 4, 5\}$



1.3 Problema SAT

Problema satisfiabilitatii booleene (**SAT**)[2] intreaba daca o formula in **forma normal conjunctiva**[3] este *satisfiabila* (daca exista o interpretare care satisface formula).

Atlfel spus, intreaba daca exista o alegere de valori (true/false) pentru variabilele (literalii) unei formule booleene, astfel incat formula sa se evalueze la true (sa fie adevarata).

2 Alogritmi in timp exponential

In continuare, se vor prezenta doua abordari de a rezolva problema kClique in timp *exponential.*

2.1 Generarea tuturor subgrafurilor de dimensiune k

O varianta la indemana de a determina daca un graf contine o clica de dimensiune k este sa generam toate subgrafurile de ordin k. Asadar, pentru un graf de ordin n trebuie sa generam C_n^k subgrafuri. Apoi, fiecare subgraf este testat pentru a vedea daca formeaza o clica.

Se poate arata usor ca operatiile de generare sunt de ordinul $O(n^k)$[4], iar cele de verificare pot fi privite ca $O(k^2)$, deoarece un graf complet de dimensiune k are C_k^2 muchii. Asadar, tot procesul va rula in $\mathbf{O(n^k k^2)}$ deci, in timp *polinomial.*

[1] https://en.wikipedia.org/wiki/Clique_problem

[2] https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[3] https://en.wikipedia.org/wiki/Conjunctive_normal_form

[4] https://proofwiki.org/wiki/N_Choose_k_is_not_greater_than_n%5Ek

Observatie: Am incadrat acest algoritm in sectiunea de timp exponential, deoarece pentru k *variabil*, algoritmul va rula in timp *exponential*.[5]. In cazul in care fixam k, problema nu mai este NP-completa, deci reducerea nu ar mai avea sens.

Functia care verifica daca nodurile primite ca parametru formeaza o clica:

```python
"""
    This method is used for checking if a given set of nodes
    are forming a clique in a given graph G
"""
def is_clique(nodes, G):
    for u in nodes:
        for v in nodes:
            if u != v and v not in G[u.index - 1].neighbors:
                return False
    return True
```

Functia care genereaza subgrafuri de dimensiune k:

```python
""" This method is used for generating all combinations of nodes of length k """
def generate_all_subsets(k, G):
    # Generate all k-combinations
    subsets = list(itertools.combinations(G, k))
    subsets = [list(node) for node in subsets]  # Convert from list of tuples to list of lists

    # Check if a subset is forming a clique
    for subset in subsets:
        if (is_clique(subset, G)):
            return True

    return False
```

In continuare, prezentam si un algoritm care foloseste *backtracking-ul* si *recursivitatea*.

2.2 Backtracking & Recursivitate

Algoritmul *Bron-Kerbosch*[6] poate fi un bun candidat pentru aceasta sectiune.

Are ca scop determinarea tuturor *clicilor maximale*[7] dintr-un graf.

Complexitatea algoritmului este de $O(3^{n/3})$, insa pentru mai multe detalii, puteti consulta urmatorul link[8]

Implementarea algoritmului Bron-Kerbosch:

```python
max_clique_dim = 0
def find_cliques(potential_clique=[], remaining_nodes=[], skip_nodes=[]):
    global max_clique_dim

    if (len(remaining_nodes) == 0 and len(skip_nodes) == 0):
        max_clique_dim = max(max_clique_dim, len(potential_clique))
        return

    for node in remaining_nodes:
        new_potential_clique = potential_clique + [node]
        new_remaining_nodes = intersection(remaining_nodes, node.neighbors)
        new_skip_list = intersection(skip_nodes, node.neighbors)
        find_cliques(new_potential_clique, new_remaining_nodes, new_skip_list)

        remaining_nodes.remove(node)
        skip_nodes.append(node)
```

[5] https://en.wikipedia.org/wiki/Clique_problem#Cliques_of_fixed_size

[6] https://tinyurl.com/ycks5kp9

[7] https://math.stackexchange.com/questions/758263/whats-maximal-clique

[8] https://en.wikipedia.org/wiki/Clique_problem#Listing_all_maximal_cliques

3 Reducere polinomiala la SAT

3.1 Transformare

Fie graful $G = (V, E)$, unde $V = \{v_1, v_2, \ldots, v_n\}$ $(n \overset{\text{not}}{=} |V|)$, iar $E = \{e_1, e_2, \ldots, e_m\}$
Pentru a reduce polinomial problema kClique la SAT, vom considera:

- Clica de k noduri ca fiind un vector de **sloturi** de dimensiune k: $[s_1, s_2, \ldots, s_k]$
- Variabilele cu care vom construi formula pentru SAT vor fi de forma x_{ij}. Literalul poate fi interpretat astfel: **nodul i ocupa slotul j (in clica)**

Inainte de a incepe reducerea propriu-zisa, mentionam faptul ca vor fi necesare $k \cdot |V|$ variabile de forma x_{ij}. In cele ce urmeaza, vom elabora formula care va constitui input pentru problema SAT.
Primele 3 constrangeri tin de logica sloturilor, iar ultima este specifica problemei kClique.

3.1.1 Fiecare slot sa fie ocupat

Construim clauze de forma: $\boxed{\bigvee_{j \in k} x_{ij},\ i = \overline{1, n}}$

3.1.2 In fiecare slot, doar o variabila are voie sa fie true

Nu ne dorim ca intr-un slot mai multe variabile sa fie true.
Deci, putem pune conditiile astfel: $\overline{x_{ij} \wedge x_{hj}}$, $\forall\, j = \overline{1, k}$ si $1 \le i < h \le n$

Aplicand legea lui De Morgan[9], obtinem clauze de forma: $\boxed{\overline{x}_{ij} \vee \overline{x}_{hj},\ \forall\, j = \overline{1, k} \text{ si } 1 \le i < h \le n}$

3.1.3 Un nod nu are voie sa fie in 2 sloturi simultan

Nu ne dorim ca un nod sa se afle in 2 sloturi in acelasi timp.
Deci, putem pune conditiile astfel: $\overline{x_{ij} \wedge x_{ih}}$, $\forall\, i = \overline{1, n}$ si $1 \le j < h \le k$

Analog, aplicand De Morgan, obtinem clauzele: $\boxed{\overline{x}_{ij} \vee \overline{x}_{ih},\ \forall\, i = \overline{1, n} \text{ si } 1 \le j < h \le k}$

3.1.4 Oricare 2 noduri din clica sa fie conectate

Pentru a genera clauzele dorite, putem gandi astfel: Pentru orice muchie (u, v) care **nu** apartine grafului, nici ambele noduri (u si v) nu trebuie sa apartina clicii.
Atunci, constrangerile vor fi de tipul: $\overline{x_{ui} \wedge x_{vj}}$, $1 \le i,\ j \le k$ si $\forall\, u, v \in V$, a.i. $(u, v) \notin E$

Astfel, obtinem clauze de forma: $\boxed{\overline{x}_{ui} \vee \overline{x}_{vj},\ 1 \le i,\ j \le k \text{ si } \forall\, u, v \in V, \text{ a.i. } (u, v) \notin E}$

3.2 Complexitate

In cele ce urmeaza, vom analiza complexitatea transformarii prezentate anterior. Vom discuta fiecare constrangere din punct de vedere al complexitatii temporale.
Inainte de toate, facem precizarea ca un graf cu n noduri, poate contine o clica de dimensiune k, cu k cel mult n (deci, $k \le n$; observatie: egalitatea are loc in cazul unui graf complet).

3.2.1 Fiecare slot sa fie ocupat

Se observa usor ca linia 4 se incadreaza in clasa de complexitate $O(k)$, linia 6 in $O(n)$, iar celelalte linii se executa in timp constant $O(1)$. Asadar, complexitatea totala pentru a genera aceste constrangeri este $O(k \cdot n)$. Cum $k \le n$, putem considera $\boxed{O(n^2)}$

```
1 """ Constraint 1 """
2 def each_slot_taken(k, N):
3     constraint = ""
4     for j in range(1, k + 1):
5         constraint += "("
6         for i in range(1, N + 1):
7             constraint += f"x{i}{j}"
```

[9] https://en.wikipedia.org/wiki/De_Morgan%27s_laws

```
            if i < N:
                constraint += " V "
        constraint += ")"
        if j < k:
            constraint += " ^ "

    return constraint
```

3.2.2 In fiecare slot, doar o variabila sa fie true

Se observa usor ca linia 4 se incadreaza in clasa de complexitate $O(k)$, linia 5 in $O(n)$, linia 6 in $O(n)$, iar celelalte linii se executa in timp constant $O(1)$. Asadar, complexitatea totala pentru a genera aceste constrangeri este $O(k \cdot n^2)$. Cum $k \leq n$, putem considera $\boxed{O(n^3)}$

```
1 """ Constraint 2 """
2 def each_slot_only_one_true(k, N):
3     constraint = ""
4     for j in range(1, k + 1):
5         for i in range(1, N + 1):
6             for h in range(i + 1, N + 1):
7                 constraint += f"(~x{i}{j} V ~x{h}{j}) ^ "
8 
9     return constraint[:-3]
```

3.2.3 Un nod nu are voie sa fie in 2 sloturi simultan

Se observa usor ca linia 3 se incadreaza in clasa de complexitate $O(n)$, linia 4 in $O(k)$, linia 5 in $O(k)$, iar celelalte linii se executa in timp constant $O(1)$. Asadar, complexitatea totala pentru a genera aceste constrangeri este $O(k^2 \cdot n)$. Cum $k \leq n$, putem considera $\boxed{O(n^3)}$

```
1 def each_node_only_one_slot(k, N):
2     constraint = ""
3     for i in range(1, N + 1):
4         for j in range(1, k + 1):
5             for h in range(j + 1, k + 1):
6                 constraint += f"(~x{i}{j} V ~x{i}{h}) ^ "
7 
8     return constraint[:-3]
```

3.2.4 Oricare 2 noduri din clica sa fie conectate

Se observa usor ca linia 7 se incadreaza in clasa de complexitate $O(|E|)$, linia 9 in $O(k)$, linia 10 in $O(k)$, iar celelalte linii se executa in timp constant $O(1)$. Asadar, complexitatea totala pentru a genera aceste constrangeri este $O(|E| \cdot k^2)$. Cum $k \leq n$ si $|E| \leq n^2$, putem considera $\boxed{O(n^4)}$

```
1 """ Constraint 4 """
2 def any_two_nodes_from_clique_connected(k, complement_edges):
3     constraint = ""
4     if len(complement_edges) == 0:
5         return constraint
6 
7     for idx, node in enumerate(complement_edges):
8         u, v = node
9         for i in range(1, k + 1):
10            for j in range(1, k + 1):
11                constraint += f"(~x{u}{i} V ~x{v}{j})"
12                if j < k:
13                    constraint += " ^ "
14            if i < k:
15                constraint += " ^ "
16        if idx < len(complement_edges) - 1:
17            constraint += " ^ "
18 
19    return constraint
```

3.2.5 Complexitatea totala

Considerand toate complexitatile determinate anterior, obtinem complexitatea totala pentru a reduce problema kClique la problema SAT: $O(n^2) + O(n^3) + O(n^3) + O(n^4) = O(n^4)$.

Asadar, complexitatea temporala pentru intregul algoritm este de $\boxed{O(n^4)}$, deci transformarea se executa in **timp polinomial**.

4 Comparatii

In continuare, vom analiza timpii de executie pentru cele 3 categorii de teste.
Din fisierele de input am extras urmatoarele atribute, pentru fiecare categorie in parte:

- Categoria 1: $(K, |V|, |E|) \in \{(7, 10, 40), (7, 10, 40), (7, 11, 43), (7, 13, 50)\}$
- Categoria 2: $(K, |V|, |E|) \in \{(3, 20, 54), (4, 50, 50), (3, 115, 219), (2, 150, 200), (3, 145, 277), (3, 135, 248)\}$
- Categoria 3: $(K, |V|, |E|) \in \{(4, 6, 10), (5, 6, 10), (6, 10, 20), (4, 6, 13), (5, 7, 13), (5, 7, 20), (4, 8, 15), (5, 8, 17), (6, 9, 24), (6, 10, 10)\}$

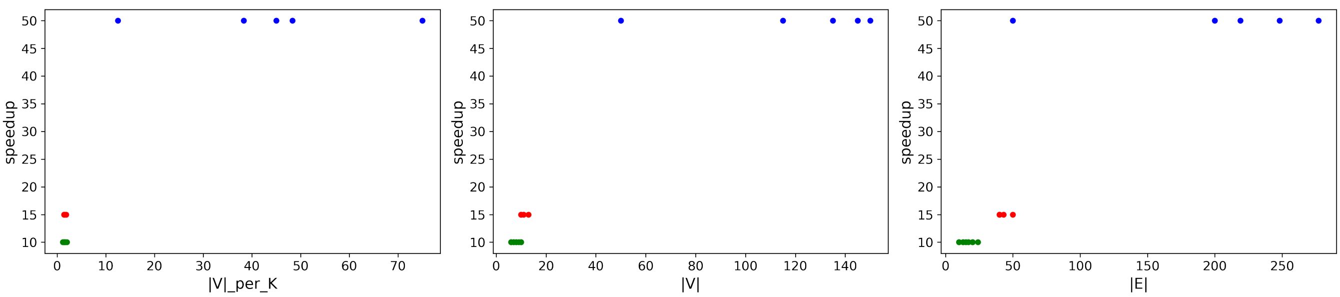
Speedup-ul masurat de checker este dat de raportul $\frac{\Delta t_{SAT}}{\Delta t_{EXP}}$

$$\begin{cases} \Delta t_{SAT} = \text{timpul necesar } \textit{reducerii + SAT solver-ului} \text{ pentru a decide problema} \\ \Delta t_{EXP} = \text{timpul necesar rularii } \textit{algoritmului exponential} \text{ pentru a decide problema} \end{cases}$$

Pentru fiecare categorie, s-au inregistrat urmatoarele speedup-uri: $\begin{cases} \text{Categoria 1: } 15 \\ \text{Categoria 2: } 50 \\ \text{Categoria 3: } 10 \end{cases}$

$$\begin{cases} \text{Speed-up } \textbf{mare} \Rightarrow \text{Preferam algoritmul } \textbf{EXP} \\ \text{Speed-up } \textbf{mic} \Rightarrow \text{Preferam reducerea la } \textbf{SAT} \end{cases}$$

Am colorat diferit fiecare categorie pentru a interpreta mai usor graficele urmatoare:



Analizand cele 3 grafice, observam ca reducerea este favorabila in cazul categoriilor 1 (rosu) si 3 (verde). In cazul categoriei 2 (albastru), reducerea + SAT solver-ul consuma mult prea mult timp, in comparatie cu algoritmul exponential, pentru a fi preferata.

Astfel, putem extrage atributele care conduc la diferitele speedup-uri:

- Cand raportul $\frac{|V|}{k}$ este **mic** (dimensiunea clicii cautate este *aproape* de $|V|$), atunci este preferata reducerea la SAT
- Cand numarul de noduri $|V|$ este **mic**, preferam reducerea
- Cand numarul de muchii $|E|$ este **mic**, preferam reducerea

Asadar, cand graful contine *multe noduri* sau *multe muchii* si dimensiunea clicii cautate (k) este *mica*, preferam sa rezolvam problema folosind direct algoritmul exponential.

Reamintim ca pentru a reduce kClique la SAT, aveam nevoie de $k \cdot |V|$ variabile (literali).

Contraintuitiv, mai multe variabile in formula data ca input SAT solver-ului, nu implica neaparat un timp de rulare mai mare.

Putem scrie cele $k \cdot |V|$ variabile necesare sub forma $|V|^2 \cdot \frac{k}{|V|}$. Din grafice, am vazut ca daca raportul $\frac{|V|}{k}$ este mic, reducerea la SAT este o varianta buna de a rezolva problema kClique.

Raportul $\frac{|V|}{k}$ mic implica $\frac{k}{|V|}$ mare. Cum $k \leq |V|$, rezulta ca vor fi necesare cel mult $|V|^2$ variabile. Insa, cu cat raportul $\frac{k}{|V|}$ este mai aproape de 1 (cu cat avem mai multi literali), cu atat putem spune ca reducerea la SAT este favorabila (putin contra intuitiei).

O analiza mai complexa pe baza atributelor extrase se poate gasi in acest Jupyter Notebook: