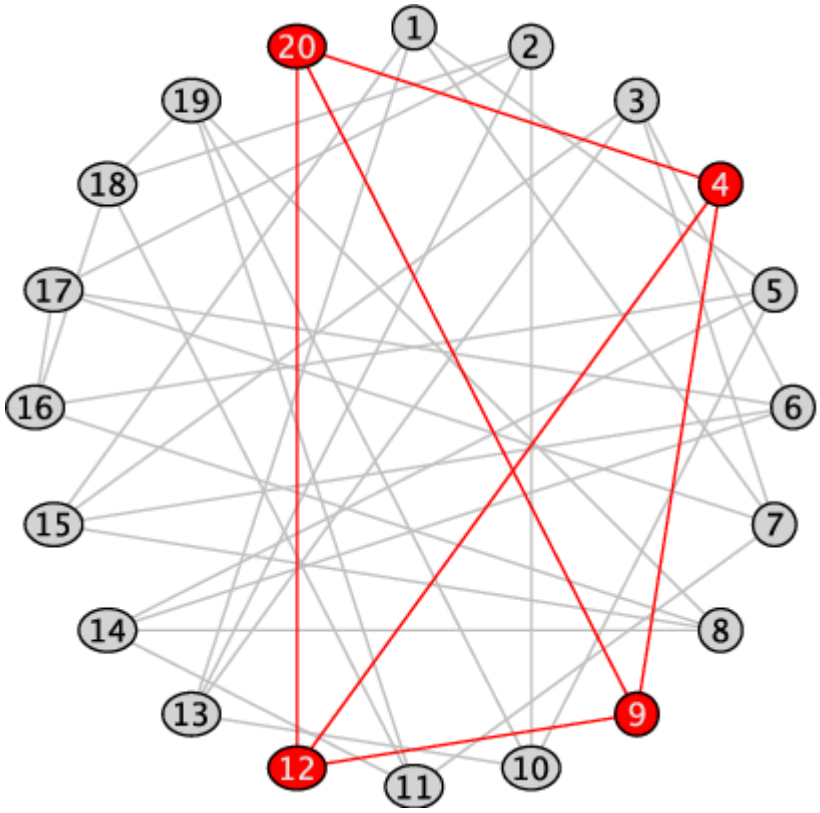
Problema k-Clique :

O clica a unui graf este un subset al varfurilor sale care sunt toate reciproc conectate. Mai jos este reprezentarea vizuala a unui grafic cu 20 de varfuri.



Desi graficul contine o clica de dimensiunea k, nu este atat de usor sa il gasim doar prin a ne uita la grafic.

In general, determinarea daca un grafic contine o clica de dimensiune k este o problema NP, adica este usor de verificat corectudinea unei solutii daca unul poate fi gasit. Este de asemenea, NP-completa cel mai dificil tip de probleme din NP. Din ce am citit si gasit pe net o astfel de problema poate fi rezolvata folosind FindClique cu ajutorul la NetworkX, dar bineinteles nu e acesta scopul temei, asa ca vom rezolva aceasta problema prin codificarea ei in logica booleana si folosind ajutorul unui SAT solver. Un SAT solver accepta o formula in logica booleana si returneaza o atribuire satisfacatoare (daca exista una).

Pentru a arata ca k-Clique se poate reduce la SAT presupunem ca avem un graf G = (V, E) si un numar k, o sa avem variabile x_{iv} pentru fiecare $1 \leq i \leq k$ si fiecare $v \in V$. O sa ne gandim la x_{iv} ca afirmatie ca v este al i-lea varf din clica, adica vom avea $k|V|$ variabile $x_{iv.}$ Vom avea urmatoarele restrangeri :

1. Pentru fiecare i, exista un i-lea varf din clica : $V_{v \in V} x_{iv}$

2. Pentru fiecare i, j al i-lea varf este diferit de al j-lea : pentru fiecare $1 \leq i \leq j \leq k$ si $v \in V$, $\neg x_{iv} \vee \neg x_{jv}$.

3. Orice doua varfuri din clica sunt conectate : pentru fiecare $1 \leq i \leq j \leq k$ si v, u $\in V$ astfel incat $(v, u) \notin E$, $\neg x_{iv} \vee \neg x_{ju}$.

Restrangeri + explicarea lor si de ce le-am folosit :

- Vom folosi variabilele booleene x_i unde i este un varf a lui G pentru a reprezenta daca varful i apare in clica cu marimea k dorita.
- Trebuie sa aplicam o constrangere care sa ne spuna daca x_i si x_j sunt adevarate (pentru orice varfuri $1 \leq i , j \leq n$) atuncia muchia {i, j} exista in graficul G.
- In mod echivalent, daca muchia {i, j} nu exista in graficul G, atunci variabilele x_i si x_j nu au cum sa fie ambele adevarate (pentru orice varf i, j). In alte cuvinte, daca {i, j} este o margine a complementului lui G, atunci stim sigur clauza $\neg x_i \vee \neg x_j$.

Constrangeri la marime:

- Avem nevoie de o modalitate pentru a impune ca clica gasita este de dimensiunea k. Cel mai naiv mod pentru a encoda asa ceva este: $(x_1 \wedge \ldots \wedge x_k) \vee \ldots \vee (x_{n-k+1} \wedge \ldots \wedge x_n)$, dar acest mod este foarte ineficient.

O codificare mai inteligenta ar fi sa folosim booleene de contor $s_{i,j}$ pentru codificare, unde $0 \leq i \leq n$ si $0 \leq j \leq k$ care cel putin j din variabilele din x_1 , x_2 , … x_i sunt asignate ca fiind true.

- Stim ca $s_{0,j}$ o sa fie false pentru $1 \leq j \leq k$ si ca $s_{i,0}$ o sa fie true pentru $0 \leq i \leq n$.
- De asemenea, stim ca $s_{i,j}$ este true daca si numai daca $s_{i-1,j}$ este true sau x_i este true si $s_{i-1,j-1}$ este true. Putem reprezenta asta prin clauza asa :

$S_{i,j} \Leftrightarrow (s_{i-1,j} \vee (x_i \wedge s_{i-1,j-1}))$ pentru $1 \leq i \leq n$ si $1 \leq j \leq k$.

- Pentru a impune ca clica gasita contine cel putin un varf k, atunci asignam si $s_{n,k}$ ca true.

Time complexity :

Problema k-Clique este o problema foarte importanta din NP-complete. Cand k este o constanta fixa, cel mai rapid algoritm cunoscut din punct de vedere asimptotic pentru gasirea unei k-clice intr-un grafic cu n-noduri ruleaza in $O(n^{792k})$ timp (dat de Nesetril si Poljak). Cu toate acestea, acest algoritm este inaplicabil, deoarece se bazeaza pe multiplicarea rapida a matricei a lui Coppersmith si Winograd.

Putem exprima k-Clique ca o instanta SAT cu O(nk) variabile si $O(nk^2)$ clauze. Pentru k fix, aceste este liniar in n. Fie $x_{iv} = 1$ daca v este al i-lea varf din clica, in alte cuvinte x_i este encodarea a i-lui varf din clica (este vectorul caracteristic pentru o multime cu un element). Aceasta introduce nk variabile.

Pentru fiecare (i, j) putem impune ca cele doua varfuri corespunzatoare sunt conectate printr-o muchie, folosind n clauze. De exemplu, o clauza este $(\neg x_{iu} \vee x_{jv1} \vee \ldots \vee x_{jvm})$ unde $v_1, v_2, \ldots, v_m$ sunt varfurile care sunt adiacente varfului u. Pentru fiecare varf u primim o clauza. Aceasta introduce un total de $O(nk^2)$ clauze.

Stiind ca sunt k clauze in φ, punem intrebarea daca graficul are o k-clica. Pretentia este ca o astfel de clica exista in G daca si numai daca exista o atribuire satisfacatoare pentru φ :

Daca directia (daca exista clica in G, atunci exista o atribuire satisfacatoare in φ):

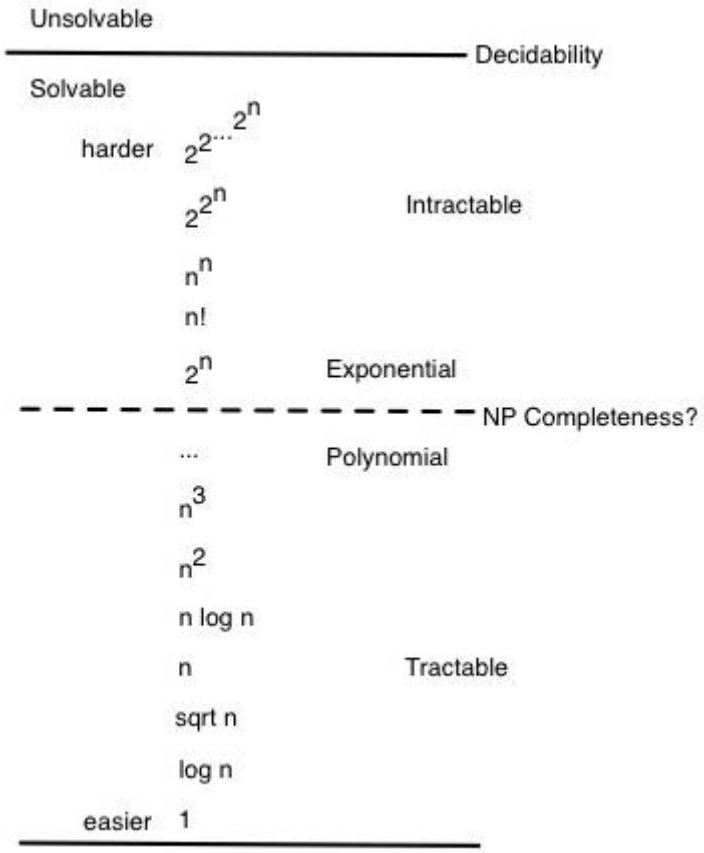
- Prin definitia k-clicii, existenta unei k-clici implica ca exista k varfuri in G care sunt toate conectate intre ele.
- Prin constructia noastra, faptul ca doua varfuri sunt conectate unul la celalalt inseamna ca pot primi o atribuire booleana consistenta (putem atribui 1 tuturor) si ca sunt in clauze diferite.
- Deoarece exista k varfuri in clica, atunci cel putin unui literal din fiecare k clauze i-se poate atribui un 1, adica poate fi satisfacuta formula φ.

Numai daca directia (daca φ poate fi satisfacuta, atunci exista o clica in G) :

- Daca φ poate fi satisfacut, atunci putem atribui valori literalilor, astfel incat cel putin unui literal din fiecare clauza I se atribuie valoarea 1, adica sunt consistente.

- Luam in considerare varfurile corespunzatoare acelor literali, deoarece literalii sunt consecventi si sunt in clauze diferite, exista o margine intre fiecare pereche din ele.
- Deoarece exista k clauze in φ avem un subset de cel putin k varfuri in grafic cu muchii intre fiecare pereche de varfuri, adica avem o k-clica in G.

Astfel daca putem rezolva k-Clique atunci putem rezolva si orice instanta de SAT, si pentru ca stim ca SAT este NPC si tranzitiva, putem rezolva orice instanta in NP in timp polinomial.



NP desemneaza clasa de probleme pentru care solutiile sunt verificabile in timp polinomial – avand in vedre o descriere a problemei x si un “certificat” y care descrie o solutie (sau care ofera suficiente informatii pentru a arata ca exista o solutie) care este polinomiala in dimensiunea x, putem verifica solutia (sau verificam ca exista o solutie) in timp polinomial in functie de x si y.

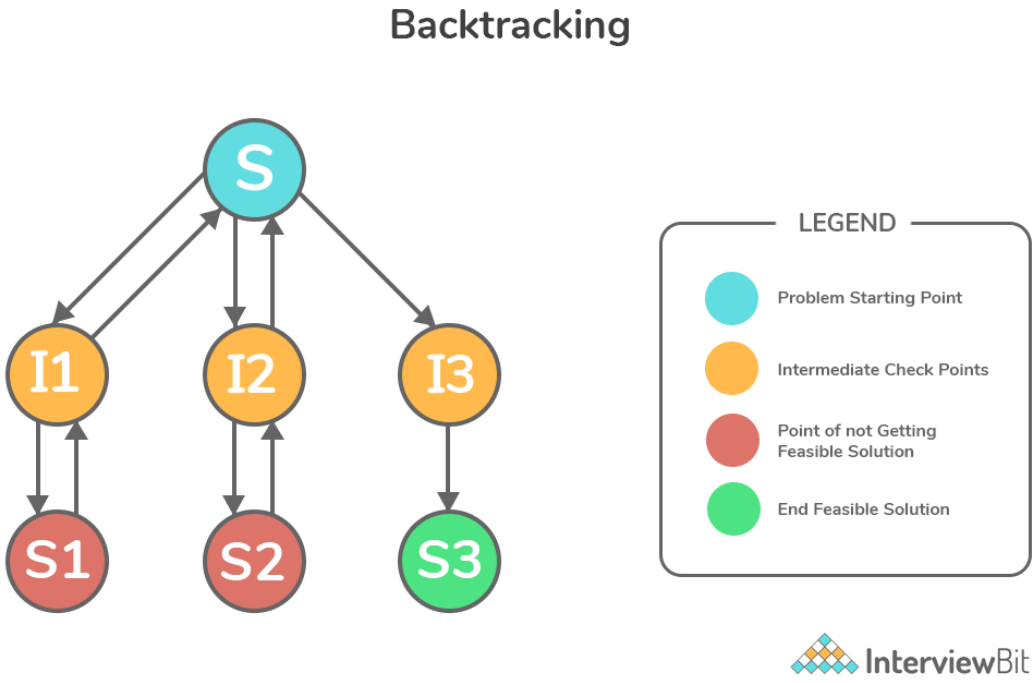
Comparare timp de executie backtracking vs reducere + SAT solver :

```
RUNNING BACKTRACKING category1
[TEST0] - PASSED
[TEST1] - PASSED
[TEST2] - PASSED
[TEST3] - PASSED
TOTAL: 4/4
BACKTRACKING TOTAL TIME: 0.067s

RUNNING REDUCTION category1
[TEST0] - PASSED
[TEST1] - PASSED
[TEST2] - PASSED
[TEST3] - PASSED
TOTAL: 4/4
REDUCTION TOTAL TIME: 1.210s

REDUCTION / BACKTRACKING: 18.059
```

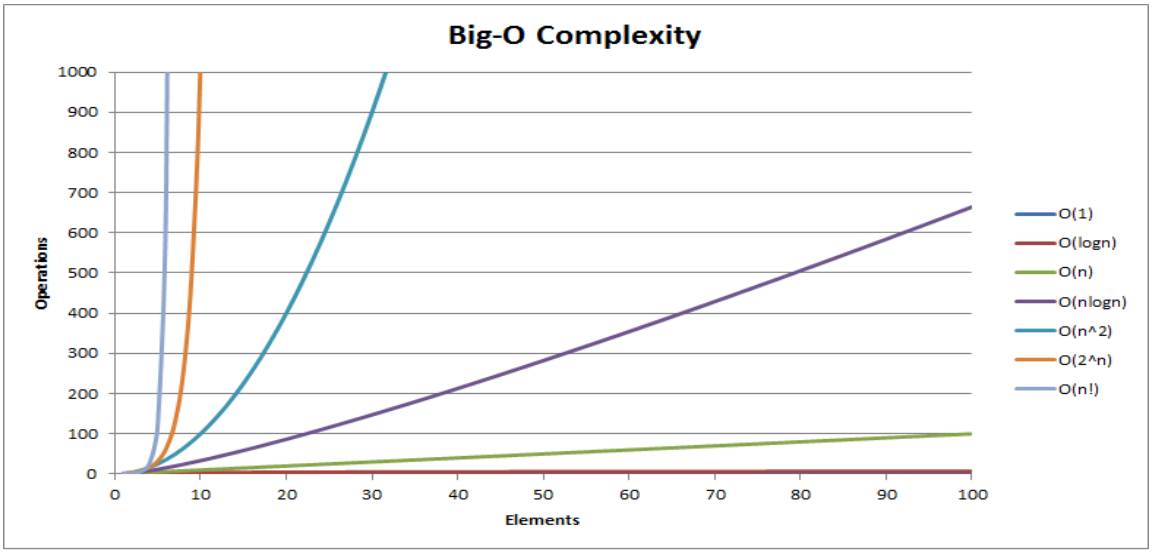
```
RUNNING BACKTRACKING category3
[TEST0] - PASSED
[TEST1] - PASSED
[TEST2] - PASSED
[TEST3] - PASSED
[TEST4] - PASSED
[TEST5] - PASSED
[TEST6] - PASSED
[TEST7] - PASSED
[TEST8] - PASSED
[TEST9] - PASSED
[TEST10] - PASSED
TOTAL: 11/11
BACKTRACKING TOTAL TIME: 0.128s

RUNNING REDUCTION category3
[TEST0] - PASSED
[TEST1] - PASSED
[TEST2] - PASSED
[TEST3] - PASSED
[TEST4] - PASSED
[TEST5] - PASSED
[TEST6] - PASSED
[TEST7] - PASSED
[TEST8] - PASSED
[TEST9] - PASSED
[TEST10] - PASSED
TOTAL: 11/11
REDUCTION TOTAL TIME: 2.204s

REDUCTION / BACKTRACKING: 17.218
```

Pentru category 1 si 3, timpurile de executie sunt relativ similare daca se ia in calcul faptul ca cateogry3 are mai multe teste de rulat. Motivul pentru

care algoritmul de backtracking ruleaza considerabil mai rapid e datorita optimizarilor facute si explicate in README. Cum implica si numele, o optimizare facuta este ca atunci cand facem verificarea pentru noduri, se va face doar de la nodurile ramase dincolo de nodul curent. Logica este explicata si in desenul de mai jos.



Desi o functie exponentiala creste mai rapid decat una polinomiala, trebuie sa ne uitam mai atent la cod pentru a vedea exact de ce BKT care e exponentiala ruleaza mai rapid decat Reduction care e polinomiala.



```
for (int i = 0; i < K; i++)
    for (int j = i + 1; j < K; j++)
        for (int v = 0; v < N; v++)
            printf("^(~x_%d_%dV~x_%d_%d)", i + 1, v + 1, j + 1, v + 1);
```

```
for (int u = 0; u < N; u++)
    for (int v = 0; v < N; v++)
        if (u != v && G[u][v] == 0)
        {
            for (int i = 0; i < K; i++)
                for (int j = 0; j < K; j++)
                    if (i != j)
                        printf("^(~x_%d_%dV~x_%d_%d)", i + 1, u + 1, j + 1, v + 1);
        }
```

Mai sus am atatsat niste snippeturi de cod, din ce putem vedea este ca implementarea pentru Reduction are foarte multe nested for-loops. Pentru primul snippet executia ar dura $O(n^3)$ timp, iar pentru a doua ar dura $O(n^4)$. Cu cat e numarul de noduri si muchii mai mare cu atat va creste timpul de executie. De asemenea deoarece avem k individual, la cresterea lui k timpul de executie o sa fie afectat foarte rau. Un test ar fi modificare la category2 la test2.in in loc de k = 3 sa punem k = 6, si timpul de executie este aproximativ de 3 ori mai mare. Astfel, deoarece trebuie sa generam toate clauzele si trebuie sa verificam pentru fiecare i < k si j < k si dupa care sa mai facem inca o verificare pentru u < N si v < N, timpul de executie va fi mereu mai prost pentru Reduction fata de BKT. Nu am sa mai trec prin de ce algoritmul de BKT este eficient deoarece este explicat in README, si aici e suficient doar sa explic de ce Reduction ruleaza atat de lent.

```
RUNNING REDUCTION category2
[TEST0] - PASSED
[TEST1] - PASSED
[TEST2] - PASSED
[TEST3] - PASSED
[TEST4] - PASSED
[TEST5] - PASSED
TOTAL: 6/6
REDUCTION TOTAL TIME: 41.934s
```

Category 2 are cel mai prost timp de executie din toate categorile, pentru a ne de seama de ce dureaza atat de mult vom face niste comparatii.

In stanga avem testul 3 din category2, iar in dreapta avem testul0 tot din category2. La rulare testul0 ruleaza aproximativ de 8 ori mai rapid ca testul 3, desi testul 0 cauta o k-clica de dimensiune 3 care este mai mare decat k-clica de dimensiune 2 cautat de testul3. Cum a fost explicat si mai sus vom avea $O(nk^2)$ clauze, adica cu cat e k mai mare cu atat timpul de rulare o sa dureze mai mult, dar in cazul nostru testul cu clica mai mare ruleaza mai rapid ca testul cu clica de dimensiune mai mica. Asta se datoreaza din cauza numarului foarte mare de noduri si muchii din graf, si deoarece algoritmul implementat la Reduction trece prin toata clauzele si verifica toate perechile de varfuri neconectate, cu cat creste mai mult numarul de noduri si muchii cu atat o sa creasca si timpul de executie. In caz contrar, la backtracking oricat de mare ar fi numarul de muchii / noduri sau oricat de mare ar fi dimensiunea k-clicii care doreste a fi gasita, timpul de executie este afectat foarte putin. Asta este din cauza faptului ca asa a fost implementat algoritmul (se poate verifica in README pentru explicatii detaliate despre cum functioneaza acesta), pe scurt el iese din etapele recursive si nu pierde timp ca in cazul la Reduction unde trece prin fiecare etapa de recursivitate.

References :

1. Cook, S. A. (1971), ”The complexity of theorem-proving procedures”, Proc. 3rd ACM Symposium on Theory of Computing
2. Karp, Richard M. (1972), ”Reducibility among combinatorial problems”, in Miller, R. E.; Thatcher, J. W., Complexity of Computer Computations, New York: Plenum
3. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. J. Symbolic Computation
4. J. Chen, X. Huang, I. A. Kanj, and G. Xia. Linear FPT reductions and computational lower bounds. In Proc. STOC
5. F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. Theor. Comp. Sci.
6. G. J. Woeginger. Space and time complexity of exact algorithms: Some open problems. In Proc. IWPEC, LNCS 3162