Εφαρμοσμένη Συνδυαστική

Backtracking

Κώστας Μανές

Τμήμα Πληροφορικής, Πανεπιστήμιο Πειραιώς

2022-2023

Backtracking

Η μέθοδος backtracking εξετάζει εξαντλητικά τον χώρο (search space) όλων των υποψήφιων λύσεων. Συνήθως κάθε υποψήφια λύση εκφράζεται ως μια λέξη $x = x_1 x_2 \cdots x_n$ σε κάποιο αλφάβητο $A = A_1 \times A_2 \times \cdots \times A_n$ (οι λύσεις έχουν συνήθως αλλά όχι πάντα ίδιο μήκος ως λέξεις). Η x είναι λύση όταν ικανοποιεί κάποιο κατηγόρημα P (δηλαδή, $P(x)$ αληθές ανν x λύση). Έτσι, ο χώρος αναζήτησης είναι το A και το σύνολο των λύσεων είναι το $S = \{x \in A : P(x)\}$. Επεκτείνοντας το P σε ένα κατηγόρημα P_k για τα προθέματα μήκους k, για κάθε k με $0 \leq k \leq n$, τέτοιο ώστε $P(x_1 \cdots x_k) \Rightarrow P_k(x_1 \cdots x_k)$ και

$$(\exists y \in A_{k+1}, P_{k+1}(x_1 \cdots x_k y)) \Rightarrow P_k(x_1 \cdots x_k)$$

(δηλαδή ένα μη αποδεκτό πρόθεμα μήκους k δεν μπορεί να επεκταθεί σε αποδεκτό πρόθεμα μήκους $k+1$ και τελικά σε μια αποδεκτή λύση) και ορίζοντας $C_{k+1}(x_1 \cdots x_k) = \{y \in A_{k+1} : P_{k+1}(x_1 \cdots x_k y)\}$ (το σύνολο γραμμάτων y που επεκτείνουν το $x_1 \cdots x_k$ σε αποδεκτό πρόθεμα), ο αλγόριθμος περιγράφεται γενικά όπως παρακάτω:

Backtracking

```
1 Back(k, x) begin
2     if P(x) then output(x);
3     compute(C_{k+1}(x));
4     foreach y ∈ C_{k+1}(x) do Back(k + 1, xy);
```

Αλγόριθμος 1: Γενικός αναδρομικός αλγόριθμος backtracking.

Η μεταβλητή k αντιπροσωπεύει το μήκος της λέξης x.

Η αναδρομή ξεκινά με αρχική κλήση $\mathrm{Back}(0, \varepsilon)$.

Η γραμμή 2 ελέγχει αν η x είναι λύση και, αν ναι, την ''τυπώνει''.

Η γραμμή 3 είναι η πιο σημαντική στον αλγόριθμο και ο υπολογισμός που εκτελεί εξαρτάται από το πρόβλημα.

Γενικά, ενδέχεται κάποια $y \in C_{k+1}(x)$ να μην οδηγήσουν τελικά σε λύση, αλλά σε αδιέξοδο, δηλαδή σε ένα πρόθεμα που δεν είναι λύση και δεν μπορεί να επεκταθεί σε λύση, οπότε ο αλγόριθμος οπισθοχωρεί (backtracks). Ένας αλγόριθμος backtracking χωρίς αδιέξοδα ονομάζεται BEST (Backtracking Ensuring Success at Terminals), οπότε κάθε φύλλο στο δένδρο της αναδρομής είναι μια λύση.

Backtracking

Ο γενικός αλγόριθμος backtracking μπορεί να διατυπωθεί και χωρίς αναδρομή, όπως παρακάτω:

```
1  Back2()begin
2      k := 1;
3      compute(C_k);
4      while k > 0 do
5          while C_k ≠ ∅ do
6              choose x_k ∈ C_k;
7              C_k := C_k \ {x_k};
8              if P(x_1 ··· x_k) then output(x_1 ··· x_k);
9              k := k + 1;
10             compute(C_k);
11         k := k − 1;
```

Αλγόριθμος 2: Γενικός επαναληπτικός αλγόριθμος backtracking.

n-queens puzzle: Να τοποθετηθούν n βασίλισσες σε μια $n \times n$ σκακιέρα ώστε να μην υπάρχουν 2 στην ίδια γραμμή, στήλη ή διαγώνιο.

Προφανώς, σε κάθε λύση, κάθε στήλη περιέχει ακριβώς μια βασίλισσα. Έστω x_k η γραμμή στην οποία βρίσκεται η βασίλισσα της k στήλης. Μια λύση $\mathrm{x} = (x_1, \ldots, x_n)$ προφανώς είναι μια μετάθεση του $[n]$, αφού κάθε γραμμή θα περιέχει τελικά ακριβώς μια βασίλισσα.
Επιπλέον, η συνθήκη

$$1 \leq i < j \leq n \Rightarrow |x_i - x_j| \neq j - i$$

εξασφαλίζει ότι δεν υπάρχουν δύο βασίλισσες στην ίδια διαγώνιο. Για την εύρεση όλων των λύσεων, αρκεί λοιπόν να κατασκευαστούν όλες οι μεταθέσεις του $[n]$ που ικανοποιούν την παραπάνω συνθήκη.

Backtracking - n-queens

Για τον αποδοτικό έλεγχο της συνθήκης, ορίζουμε τις μεταβλητές (arrays) A, B, C, με

$A[i] = \text{false} \Leftrightarrow$ η γραμμή i περιέχει βασίλισσα, $1 \leq i \leq n$

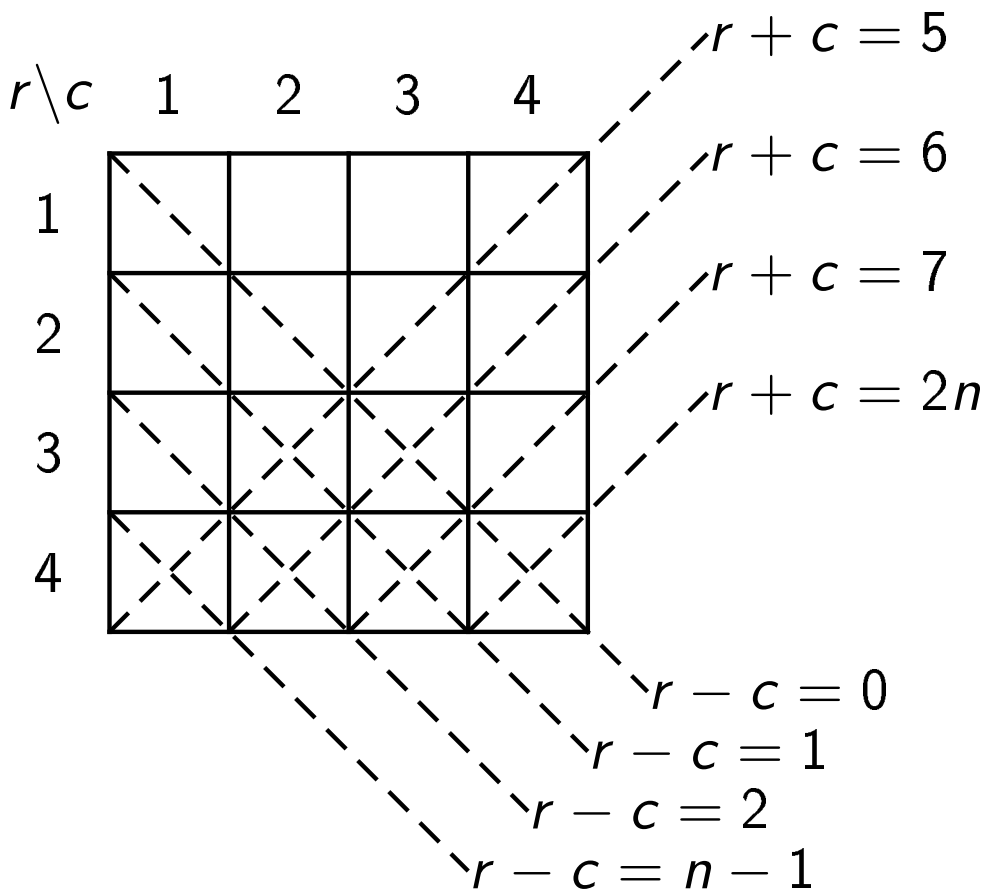
$B[i] = \text{false} \Leftrightarrow$ η διαγώνιος $r + c = i$ περιέχει βασίλισσα, $2 \leq i \leq 2n$

$C[i] = \text{false} \Leftrightarrow$ η διαγώνιος $r - c = i$ περιέχει βασίλισσα, $|i| \leq n - 1$

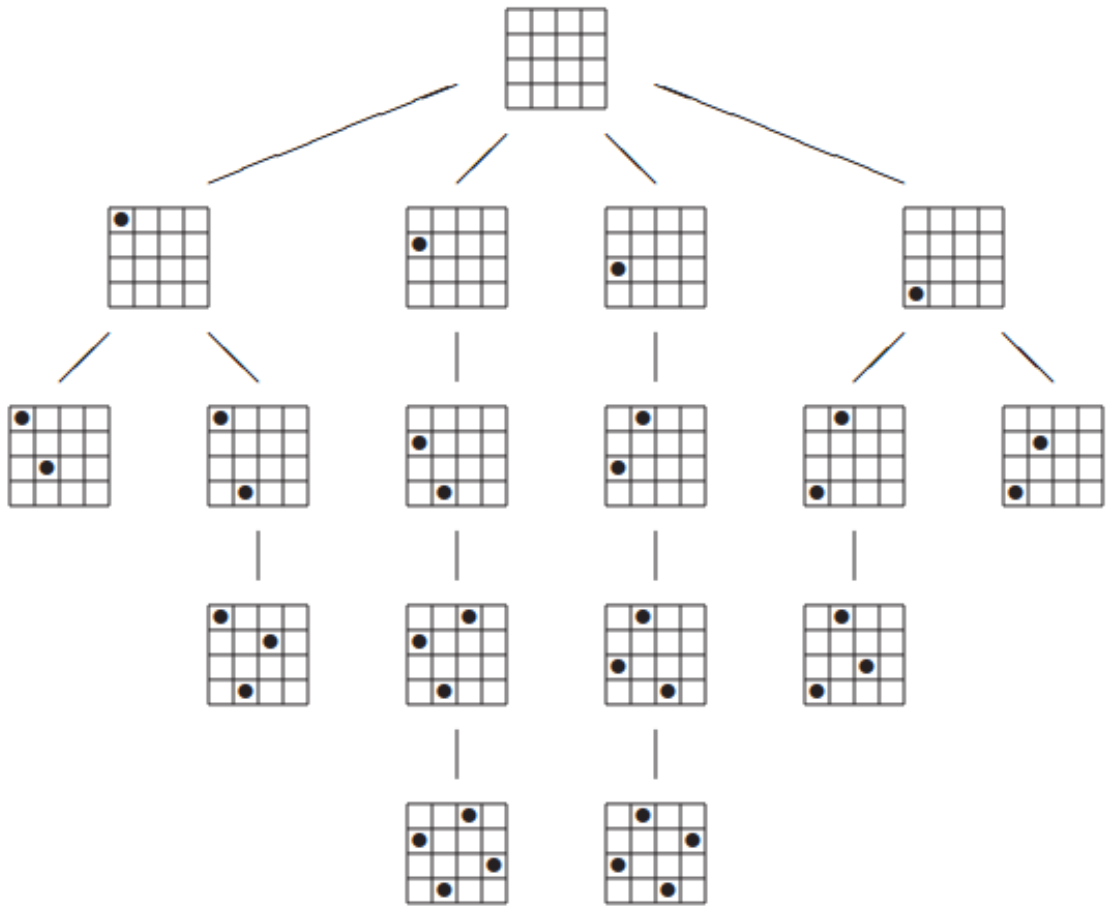
```
1 Queens(c) begin
2     for r := 1 to n do
3         if A[r] and B[r + c] and C[r − c] then
4             x_c := r;
5             A[r] := B[r + c] := C[r − c] := false;
6             if c < n then Queens(c + 1) else output(x);
7             A[r] := B[r + c] := C[r − c] := true;
```

Αλγόριθμος 3: Αλγόριθμος backtracking για το n-queens puzzle.

Backtracking n-queens



Backtracking n-queens



Σχήμα: Το δένδρο της αναδρομής για το 4-queens puzzle.

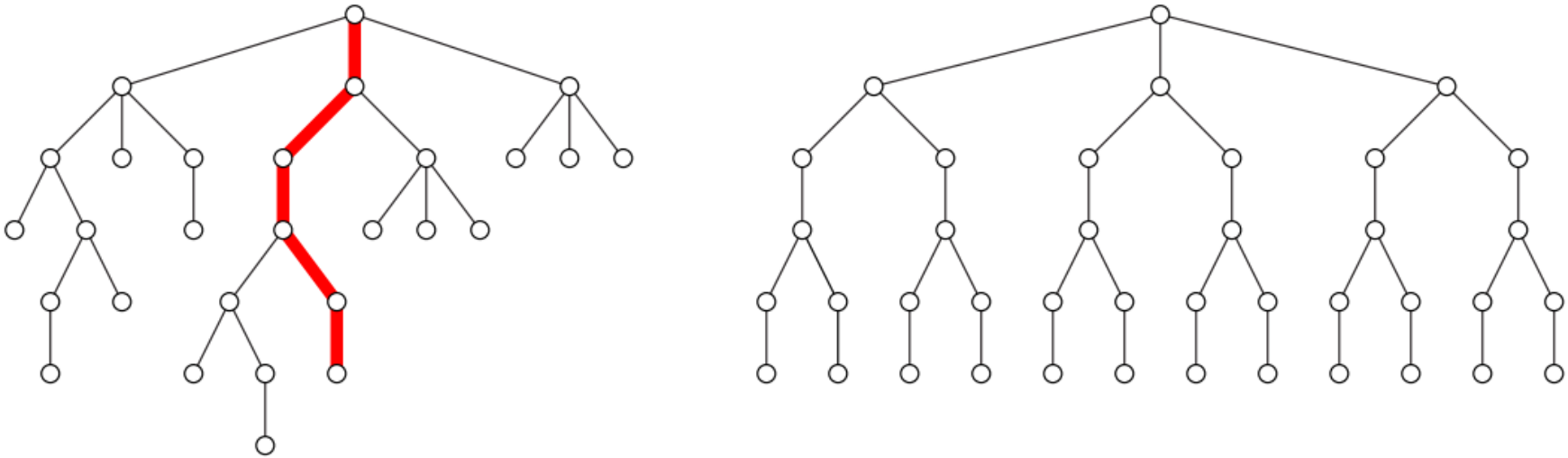
Backtracking - Εκτίμηση χώρου αναζήτησης

Για να εκτιμήσουμε το μέγεθος του χώρου αναζήτησης, δηλαδή το πλήθος κόμβων $|T|$ του δένδρου αναδρομής T, προτού εκτελέσουμε έναν αλγόριθμο backtracking, επιλέγουμε τυχαία ένα μονοπάτι $P = (x_1 = r, x_2, \ldots, x_t)$ από τη ρίζα r μέχρι κάποιο φύλλο x_t, όπου ο κόμβος x_i έχει $\mathrm{outdeg}(x_i) = n_i$ παιδιά, και θεωρούμε ότι κάθε μονοπάτι στο δένδρο έχει την ίδια ακολουθία βαθμών. Το εκτιμώμενο πλήθος κόμβων στο επίπεδο i είναι μια τυχαία μεταβλητή $X_i = n_1 n_2 \cdots n_{i-1} = X_{i-1} n_{i-1}$, $X_1 = 1$, και θέτουμε $X = \sum_{i=1}^{t} X_i$. Επιπλέον ορίζουμε για κάθε $v \in T$ την τ.μ. $I(v) = [v \in P]$, και τη συνάρτηση $\tau(v) = \mathrm{outdeg}(\bar{v})\tau(\bar{v})$, όπου $\bar{v}$ ο γονιός του v και $\tau(r) = 1$, οπότε $\tau(v) = \prod_{u \text{ πρόγονος του } v} \mathrm{outdeg}(u)$ και $X = \sum_{v \in T} \tau(v) I(v)$. Κατόπιν τούτων, η μέση τιμή της τ.μ. X ισούται με

$$E(X) = \sum_{v \in T} \tau(v) E(I(v)) = \sum_{v \in T} \tau(v) \frac{1}{\tau(v)} = |T|$$

δηλαδή η X είναι μια αμερόληπτη εκτιμήτρια του $|T|$.

Backtracking - Εκτίμηση χώρου αναζήτησης



Σχήμα: Ένα τυχαία επιλεγμένο μονοπάτι (κόκκινο) και το εκτιμώμενο δένδρο της αναδρομής (δεξιά) βάσει αυτού του μονοπατιού.

Backtracking - Εκτίμηση χώρου αναζήτησης

Επιλέγοντας αρκετά τέτοια τυχαία μονοπάτια, έχουμε τελικά μια πολύ καλή εκτίμηση για το $|T|$.

```
1  sum := 0;
2  for i := 1 to N do
3      X := product := k := 1;
4      compute(C_1);
5      while C_k ≠ ∅ do
6          n_k := |C_k|;
7          product := n_k · product;
8          X := X + product;
9          x_k := an element of C_k, chosen at random;
10         k := k + 1;
11         compute(C_k);
12     sum := sum + X;
13 output(sum/N);
```

Αλγόριθμος 4: Αλγόριθμος εκτίμησης του $|T|$.

Backtracking - Κλίκες γραφήματος

Έστω (απλό, μη κατευθυνόμενο) γράφημα $G = (V, E)$. Ένα σύνολο $C \subseteq V$ είναι κλίκα του G ανν $x, y \in C, x \neq y \Rightarrow \{x, y\} \in E$.
Ο πληθάριθμος του C είναι το μέγεθος της κλίκας.
Μια κλίκα είναι μεγιστική αν δεν είναι γνήσιο υποσύνολο άλλης κλίκας.
Μια κλίκα είναι μέγιστη αν δεν υπάρχει κλίκα με μεγαλύτερο μέγεθος.
Τετριμμένες κλίκες του G είναι το κενό σύνολο, τα μονοσύνολα και τα δισύνολα (οι ακμές) του E.
Στη συνέχεια, θα δούμε έναν αλγόριθμο backtracking για την εύρεση όλων των κλικών ενός γραφήματος G. Επίσης, θα τροποποιήσουμε τον αλγόριθμο αυτόν, ώστε να βρίσκει μια μέγιστη κλίκα του G.

Ισοδύναμα προβλήματα με αυτό της εύρεσης μέγιστης κλίκας, είναι η εύρεση μέγιστου ανεξάρτητου συνόλου (independent set) και ελάχιστου καλύμματος από κορυφές (vertex cover). Τα προβλήματα αυτά είναι NP-hard.
Υπενθυμίζεται ότι το $A \subseteq V$ είναι ανεξάρτητο σύνολο του $G = (V, E)$ ανν το A είναι κλίκα του (συμπληρώματος) G^c. Το $B \subseteq V$ είναι κάλυμμα από κορυφές ανν $\{x, y\} \in E \Rightarrow \{x, y\} \cap B \neq \varnothing$. Προφανώς, ισχύει ότι A ανεξάρτητο σύνολο ανν $V \setminus A$ κάλυμμα από κορυφές. Επομένως, A μέγιστο ανεξάρτητο σύνολο ανν $V \setminus A$ ελάχιστο κάλυμμα από κορυφές.

Για απλότητα, θεωρούμε ότι $V = \{1, 2, \ldots, n\}$, όπου $n = |V|$. Αναπαριστούμε κάθε κλίκα μεγέθους $k \in [n]$ ως μια λέξη $X_k = x_1 x_2 \cdots x_k \in V^k$. Για να αποφύγουμε τις επαναλήψεις κατά την κατασκευή, θεωρούμε ότι $x_1 < x_2 < \cdots < x_k$ (δηλαδή οι κορυφές της κλίκας καταγράφονται σε αύξουσα σειρά).
Προφανώς, κάθε υποσύνολο μιας κλίκας είναι επίσης κλίκα, οπότε η κλίκα X_k επεκτείνεται στην κλίκα $X_k y$ ανν y είναι μια κορυφή με $y > x_k$ και $\{x_i, y\} \in E$, για κάθε $i \in [k]$. Επομένως, το σύνολο επιλογών C_{k+1} (το σύνολο όλων των έγκυρων y) για την επέκταση της μερικής λύσης X_k (όπου X_0 η κενή λέξη) είναι το

$$C_{k+1} = \{y \in C_k \setminus \{x_k\} : \{x_k, y\} \in E, y > x_k\}, \qquad C_1 = V.$$

Backtracking - Κλίκες γραφήματος

Προκειμένου να υπολογίσουμε τα σύνολα C_k αποδοτικά, ορίζουμε τα σύνολα

$$N_x = \{y \in V : \{x, y\} \in E\} \quad \text{και} \quad A_x = \{y \in N_x : y > x\}, \qquad x \in V$$

των γειτόνων της κορυφής x και των γειτόνων που είναι μεγαλύτεροι της x, οπότε

$$C_{k+1} = C_k \cap A_{x_k}$$

Τα σύνολα A_x υπολογίζονται μία φορά πριν την εκκίνηση του αλγορίθμου.
Επιπλέον, για να ανιχνεύσουμε αν η X_k είναι μεγιστική, ορίζουμε το σύνολο

$$M_{k+1} = M_k \cap N_{x_k}, \qquad M_1 = V$$

των κορυφών που μπορούν να επεκτείνουν την X_k, οπότε η X_k είναι μεγιστική ανν $M_{k+1} = \varnothing$.
Κατόπιν τούτων, ο αλγόριθμος είναι ο ακόλουθος:

```
1 allCliques(k) begin                    /* global: X, C_k, A_k, N_k */
2     if k > 0 then output(x_1 x_2 ··· x_k);
3     if k = 0 then M_{k+1} := V else M_{k+1} := M_k ∩ N_{x_k};
4     if M_{k+1} = ∅ then output ("maximal");
5     if k = 0 then C_{k+1} := V else C_{k+1} := C_k ∩ A_{x_k};
6     for y ∈ C_{k+1} do
7         x_{k+1} := y;
8         allCliques(k + 1);
```

Αλγόριθμος 5: Αλγόριθμος backtracking για την εύρεση όλων των κλικών. Αρχική κλήση: allCliques(0).

Το πρόβλημα της ικανοποιησιμότητας (SATisfiability):

Δοθέντος ενός συνόλου Σ λογικών προτάσεων, να ευρεθεί εκτίμηση που να το ικανοποιεί ή η απάντηση ότι δεν υπάρχει τέτοια εκτίμηση.

Δεδομένου ότι κάθε πρόταση μπορεί να γραφεί ισοδύναμα σε CNF μορφή, το παραπάνω πρόβλημα είναι ισοδύναμο με

Το πρόβλημα SAT:

Δοθείσης μιας λογικής πρότασης ϕ σε CNF μορφή, να ευρεθεί εκτίμηση που να την ικανοποιεί ή η απάντηση ότι δεν υπάρχει τέτοια εκτίμηση.

Η μέθοδος backtracking μπορεί να ελέγξει όλες τις δυνατές εκτιμήσεις (2^n σε πλήθος όταν η ϕ περιέχει n διαφορετικές μεταβλητές), ώστε να βρει μια που ικανοποιεί την ϕ. Στη συνέχεια, θα δούμε έναν πιο γρήγορο τρόπο, μετατρέποντας το πρόβλημα αυτό σε πρόβλημα εύρεσης κλίκας με κατάλληλο μέγεθος σε γράφημα.

Το πρόβλημα 3-SAT είναι μια ειδική περίπτωση του προβλήματος SAT, όπου κάθε παρένθεση της πρότασης ϕ περιέχει ακριβώς 3 όρους. Στην περίπτωση αυτή, λέμε ότι η ϕ είναι σε μορφή 3-CNF.

Παράδειγμα

Η πρόταση

$$(p_1 \vee p_2 \vee p_3) \wedge (p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_4)$$

είναι σε μορφή 3-CNF.

Αν και το 3-SAT αποτελεί ειδική περίπτωση του SAT, τελικά είναι ισοδύναμο με το γενικό πρόβλημα SAT, διότι κάθε CNF πρόταση μετατρέπεται σε μια ισοδύναμη 3-CNF πρόταση (με έναν αλγόριθμο πολυωνυμικού χρόνου που δεν παρουσιάζεται εδώ).

Κλίκες και το πρόβλημα SAT

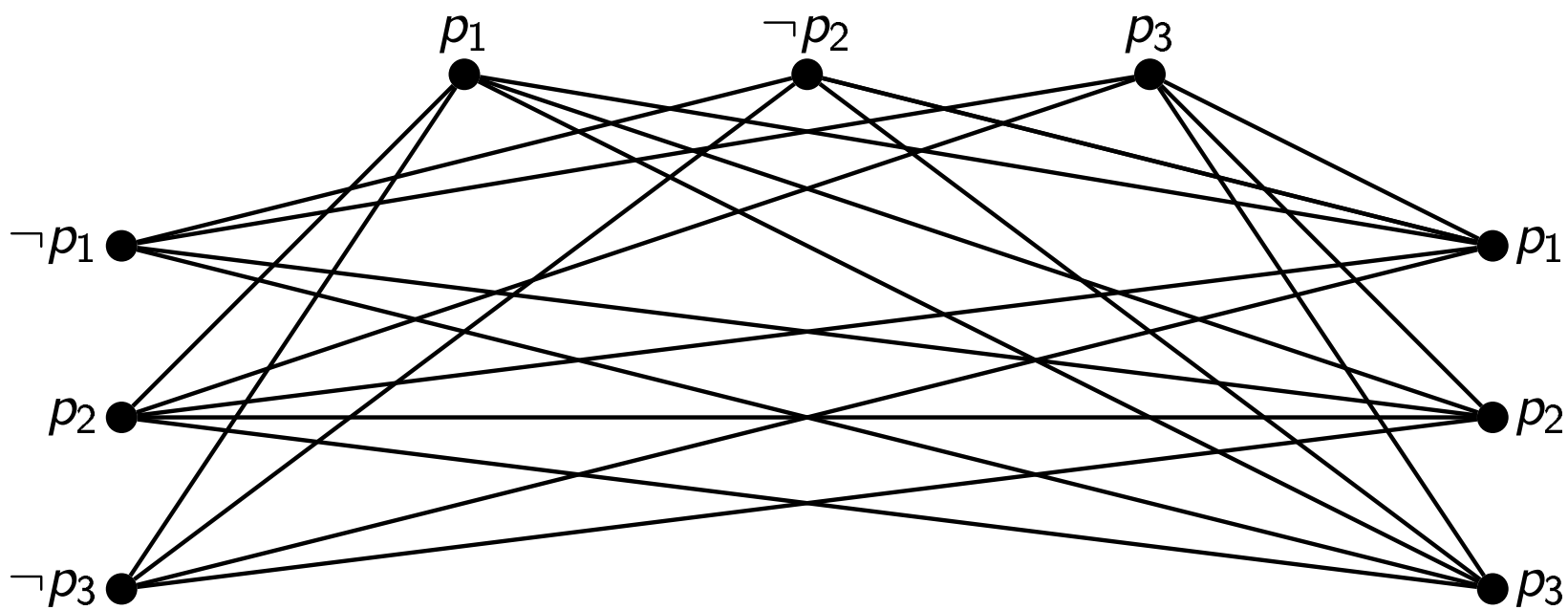
Δίνεται μια πρόταση ϕ σε μορφή 3-CNF $\phi_1 \wedge \cdots \wedge \phi_m$ και από αυτή θα κατασκευαστεί ένα γράφημα G τέτοιο ώστε κάθε κλίκα του με m στοιχεία να αντιστοιχεί σε μια εκτίμηση που ικανοποιεί την ϕ. Αν δεν υπάρχει τέτοια κλίκα, τότε η ϕ δεν είναι ικανοποιήσιμη.

- Θεωρούμε ότι το G είναι αρχικά κενό.
- Για κάθε διαζευκτικό όρο ϕ_i της ϕ, εισάγουμε 3 νέες κορυφές στο γράφημα G με ετικέτες τα ονόματα των όρων της ϕ_i. Η τριάδα αυτή αποτελεί ένα ανεξάρτητο σύνολο μεγέθους 3 στο G. Αφού κάνουμε το ίδιο για κάθε ϕ_i της ϕ, το G είναι ένα γράφημα που αποτελείται από $3m$ κορυφές και 0 ακμές.
- Τέλος, ενώνουμε κάθε κορυφή με ετικέτα p_j με κάθε άλλη κορυφή σε άλλη τριάδα και με ετικέτα διάφορη του $\neg p_j$. Έτσι, προκύπτει το τελικό γράφημα G.

Για παράδειγμα, αν

$$\phi : (\neg p_1 \vee p_2 \vee \neg p_3) \wedge (p_1 \vee \neg p_2 \vee p_3) \wedge (p_1 \vee p_2 \vee p_3),$$

Τότε το γράφημα G που προκύπτει με την παραπάνω διαδικασία είναι το ακόλουθο:



Κλίκες και το πρόβλημα SAT

- Μια κλίκα C μεγέθους m θα πρέπει να περιέχει ακριβώς έναν όρο από κάθε τριάδα.
- Αν περιέχει τη μεταβλητή p_i, δεν μπορεί να περιέχει την άρνησή της $\neg p_i$, και αντίστροφα. Στην πρώτη περίπτωση θέτουμε $v(p_i) = 1$, αλλιώς $v(p_i) = 0$.
- Σε κάθε μεταβλητή p_j που δεν εμφανίζεται η ίδια ούτε και η άρνησή της στο C, μπορούμε να δώσουμε αυθαίρετη εκτίμηση $v(p_j)$.
- Έτσι, παίρνουμε την ζητούμενη εκτίμηση v που θα ικανοποιεί την ϕ. Αν η ϕ δεν ικανοποιείται, τότε δεν υπάρχει τέτοια κλίκα C.

Backtracking - Exact Cover

Δίνεται ένα σύνολο $\mathcal{S} = \{S_0, S_1, \ldots, S_{m-1}\}$ από m υποσύνολα του $\mathcal{R} = \{0, 1, 2, \ldots, n-1\}$. Ένα υποσύνολο $\mathcal{S}' \subseteq \mathcal{S}$ τέτοιο ώστε κάθε $x \in \mathcal{R}$ περιέχεται σε ακριβώς ένα σύνολο του $\mathcal{S}'$ ονομάζεται exact cover (EC) του $\mathcal{S}$.

Αν ορίσουμε $G = (V, E)$ το γράφημα με $V = \{0, 1, 2, \ldots, m-1\}$ και $\{i, j\} \in E \Leftrightarrow S_i \cap S_j = \varnothing$, τότε ένα EC είναι μια κλίκα του G που επιπλέον καλύπτει το $\mathcal{R}$.

Προκειμένου να βρούμε όπως πριν τις κλίκες αυτές, θεωρούμε κάθε σύνολο S_i ως μια λέξη με τα στοιχεία του να είναι τα γράμματα και σε αύξουσα σειρά, οπότε τα S_i διατάσσονται σε λεξικογραφική σειρά και, χωρίς βλάβη της γενικότητας, θεωρούμε ότι
$S_0 <_L S_1 <_L \cdots <_L S_{m-1}$.

Επιπλέον, ορίζουμε δύο βοηθητικά σύνολα, το

$$A_x = \{y \in V : S_x \cap S_y = \varnothing, y > x\}$$

των γειτόνων της κορυφής x που είναι μεγαλύτεροι από αυτήν (βλ. επόμενο παράδειγμα), καθώς και το $H_i = \{j \in V : \min S_j = i\}$ και $H_n = \varnothing$.

Backtracking - Exact Cover

$\mathcal{R} = \{0, 1, 2, 3, 4, 5, 6\}$

j	S_j	A_j
0	0, 1, 3	10
1	0, 1, 5	12
2	0, 2, 4	7, 9
3	0, 2, 5	8, 9, 12
4	0, 3, 6	5, 9
5	1, 2, 4	$\varnothing$
6	1, 2, 6	11
7	1, 3, 5	$\varnothing$
8	1, 4, 6	$\varnothing$
9	1	10, 11, 12
10	2, 5, 6	$\varnothing$
11	3, 4, 5	$\varnothing$
12	3, 4, 6	$\varnothing$

i	0	1	2	3	4	5	6
H_i	0, 1, 2, 3, 4	5, 6, 7, 8, 9	10	11, 12	$\varnothing$	$\varnothing$	$\varnothing$

Backtracking - Exact Cover

```
1  ec(k, r) begin          /* global: X = x_0 ··· x_{k-1}, C_k, A_k, H_i */
2      if k = 0 then  /* U_k contains the uncovered elements at
         level k and r = min U_k */
3          U_0, C_0, r := {0, 1, ... n − 1}, {0, 1, ..., m − 1}, 0
4      else
5          U_k := U_{k-1} \ S_{x_{k-1}};
6          while r ∉ U_k and r < n do r := r + 1;
7          C_k := C_{k-1} ∩ A_{x_{k-1}}
8      if r = n then output([x_0, x_1, ..., x_{k-1}]);
9      C_k := C_k ∩ H_r;
10     for y ∈ C_k do
11         x_k := y;
12         ec(k + 1, r)
```

Αλγόριθμος 6: Αλγόριθμος backtracking για την εύρεση όλων των EC. Αρχική κλήση: ec(0,0).

Backtracking - $(0, 1)$-knapsack problem

Δίνονται n αντικείμενα, έστω τα $1, 2, \ldots, n$, καθώς και η αξία p_i και το βάρος w_i του κάθε αντικειμένου i, $i \in [n]$. Αν ένα σακίδιο παίρνει αντικείμενα συνολικού βάρους το πολύ M, ζητείται η καλύτερη επιλογή αντικειμένων που θα μπουν στο σακίδιο, δηλαδή ένα υποσύνολο αυτών με τη μέγιστη δυνατή συνολική αξία και συνολικό βάρος το πολύ M.

Χρησιμοποιώντας μια δυαδική μεταβλητή $x_i \in \{0, 1\}$ για κάθε αντικείμενο i, δηλαδή θέτοντας $x_i = 1$ αν το αντικείμενο i επιλέγεται να μπει στο σακίδιο και $x_i = 0$ αλλιώς, το πρόβλημα διατυπώνεται ως ένα πρόβλημα Ακέραιου Γραμμικού Προγραμματισμού:

$$\text{maximize } p(x_1, \ldots, x_n) := \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } w(x_1, \ldots, x_n) := \sum_{i=1}^{n} w_i x_i \leq M$$

$$x_1, x_2, \ldots, x_n \in \{0, 1\}.$$

Backtracking - $(0, 1)$-knapsack problem

Έστω S το σύνολο των λέξεων $X = x_1 x_2 \cdots x_n \in \{0, 1\}^n$ που ικανοποιούν $w(X) := \sum_{i=1}^{n} w_i x_i \leq M$. Κάθε $X \in S$ είναι μια εφικτή (feasible) λύση. Αναζητούμε εκείνη με τη μέγιστη συνολική αξία $p(X) := \sum_{i=1}^{n} p_i x_i$. Για την εύρεση αυτής, ξεκινάμε από μια αρχικά κενή λέξη X και προσθέτουμε στο τέλος αυτής το γράμμα x_{k+1} σε κάθε βήμα $0 \leq k \leq n - 1$.
Αν $w(X) \leq M$, η X επεκτείνεται στην $X0$ (δηλαδή θέτοντας $x_{k+1} = 0$), καθώς και στην $X1$, εφόσον $w(X1) = w(X) + w_{k+1} \leq M$.
Έτσι προκύπτει ο ακόλουθος αλγόριθμος:

Backtracking - $(0,1)$-knapsack problem

```
1  knapsack(k, curW) begin          /* global: X, optX, optP */
                                        /* X = (x_1, x_2, ..., x_k) */
2      if k = n then
3          if Σ_{i=1}^{k} p_i x_i > optP then        /* update optimum */
4              optP := Σ_{i=1}^{k} p_i x_i;
5              optX := X;
6      else                  /* check each choice for item k + 1 */
7          C := {0};
8          if curW + w_{k+1} ≤ M then C := {0, 1};
9          for c ∈ C do
10             x_{k+1} := c;
11             knapsack(k + 1, curW + w_{k+1} x_{k+1});
```

Αλγόριθμος 7: Αλγόριθμος backtracking για το $(0,1)$-knapsack problem. Αρχική κλήση: $\mathrm{knapsack}(0,0)$.

Backtracking - bounding

Όταν έχουμε ένα πρόβλημα βελτιστοποίησης, τότε μπορούμε να χρησιμοποιήσουμε κάποια συνάρτηση-φράγμα (bounding function) ώστε να ''κλαδέψουμε'' περαιτέρω το δένδρο της αναδρομής.
Ας υποθέσουμε ότι έχουμε ένα πρόβλημα μεγιστοποίησης και ας ορίσουμε για την μερική λύση $X_k = x_1 \cdots x_k$ να είναι $maxP(X_k)$ η μέγιστη τιμή που έχει μια αποδεκτή λύση που επεκτείνει την X_k.
Συνήθως η εκτίμηση της τιμής $maxP(X_k)$ είναι αδύνατη, οπότε χρησιμοποιούμε μια συνάρτηση B που υπολογίζεται εύκολα και ικανοποιεί $maxP(X_k) \leq B(X_k)$.
Έτσι, αν είναι $maxP(X_k) \leq B(X_k) \leq optP$, όπου $optP$ η αξία της καλύτερης λύσης που έχουμε βρει μέχρι στιγμής, αυτό σημαίνει ότι δεν χρειάζεται να εξερευνήσουμε τις επιλογές επέκτασης του X_k γιατί δεν θα οδηγήσουν σε καλύτερη λύση.
Η επιλογή της B γίνεται ανάλογα με το πρόβλημα, δεν είναι μοναδική και επιθυμούμε να προσεγγίζει όσο το δυνατόν καλύτερα την $maxP$.

Backtracking - bounding

```
1  BackB(k, x) begin
2      if P(x) then
3          if p(x) > optP then
4              optP := p(x);
5              optX := x;
6      compute(C_{k+1}(x));
7      b := B(x);
8      foreach y ∈ C_{k+1}(x) do
9          if b ≤ optP then return;
10         BackB(k + 1, xy)
```

Αλγόριθμος 8: Γενικός αναδρομικός αλγόριθμος backtracking με συνάρτηση-φράγμα B, για πρόβλημα μεγιστοποίησης.

Στον προηγούμενο αλγόριθμο, η συνθήκη στη γραμμή 8 μπορεί να γίνει αληθής σε κάποια μεταγενέστερη επανάληψη του βρόχου, καθώς η τιμή $optP$ ενημερώνεται διαρκώς. Τότε, ο βρόχος σταματά να εκτελείται.
Συνήθως, ο βρόχος της γραμμής 7 ελέγχει πρώτα τα y που δίνουν τη μεγαλύτερη τιμή $B(xy)$. Αυτό προϋποθέτει ότι τα y αποθηκεύονται σε μια ουρά προτεραιότητας. Η προσέγγιση αυτή συνήθως οδηγεί σε μεγαλύτερο κλάδεμα του δένδρου αναδρομής, γιατί η τιμή $optP$ ενημερώνεται νωρίτερα. Η μέθοδος αυτή είναι γνωστή ως Branch and Bound και παρουσιάζεται σε μια γενική μορφή στον επόμενο αλγόριθμο:

Branch and Bound

```
1 BnB()begin                                   /* global:  optX, optP */
2     Q.push(B(ε), ε);
3     while Q is not empty do
4         (b, x) = Q.pop();
5         if P(x) and p(x) > optP then (optP, optX) := (p(x), x) ;
6         compute(C_{k+1}(x));
7         foreach y ∈ C_{k+1}(x) do
8             if B(xy) > optP then Q.push(B(xy), xy) ;
```

Αλγόριθμος 9: Γενικός αλγόριθμος Branch and Bound με συνάρτηση-φράγμα B, για πρόβλημα μεγιστοποίησης.

Η μεταβλητή Q είναι μια αρχικά κενή ουρά προτεραιότητας που πάντα επιστρέφει (pop) το πρόθεμα x με την μεγαλύτερη τιμή $B(x)$. Η βέλτιστη λύση αποθηκεύεται στην σφαιρική μεταβλητή $optX$.

Συνήθως, οι μεταβλητές $optP$, $optX$ αρχικοποιούνται με κάποια υποψήφια λύση που υπολογίζεται γρήγορα, π.χ. με κάποιον άπληστο αλγόριθμο.

Backtracking - bounding - $(0,1)$-knapsack problem

Για το πρόβλημα $(0,1)$-knapsack, δοθείσης μιας μερικής λύσης $X_k = x_1 \cdots x_k$, βάρους $w(X_k) = \sum_{i=1}^{k} w_i x_i$ και αξίας $p(X_k) = \sum_{i=1}^{k} p_i x_i$, προκειμένου να πάρουμε ένα άνω φράγμα για το $maxP(X_k)$, εργαζόμαστε ως εξής: Θεωρούμε ότι τα στοιχεία είναι ταξινομημένα έτσι ώστε $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$. Αν $X_n = x_1 \cdots x_k \cdots x_n$ είναι μια επέκταση της X_k, τότε

$$p(X_n) = \sum_{i=1}^{n} p_i x_i = p(X_k) + \sum_{i=k+1}^{n} p_i x_i = p(X_k) + \sum_{i=k+1}^{n} \frac{p_i}{w_i} w_i x_i$$

$$\leq p(X_k) + \frac{p_{k+1}}{w_{k+1}} \sum_{i=k+1}^{n} w_i x_i \leq p(X_k) + \frac{p_{k+1}}{w_{k+1}}(M - w(X_k)),$$

οπότε θέτοντας

$$B(X_k) := p(X_k) + \frac{p_{k+1}}{w_{k+1}}(M - w(X_k))$$

έχουμε ότι $maxP(X_k) \leq B(X_k)$.

Backtracking - bounding - $(0,1)$-knapsack problem

Το προηγούμενο φράγμα έχει το μειονέκτημα ότι αυξάνει αναλογικά με το M, το οποίο μπορεί να είναι αρκετά μεγάλο σε κάποια στιγμιότυπα του προβλήματος. Ένα καλύτερο φράγμα μπορεί να επιτευχθεί χρησιμοποιώντας τη λύση του (κλασματικού) fractional knapsack problem, στο οποίο τα x_i παίρνουν πραγματικές τιμές στο διάστημα $[0,1]$, οπότε η λύση του προκύπτει πολύ εύκολα χρησιμοποιώντας άπληστο αλγόριθμο: Εντοπίζουμε τον ελάχιστο δείκτη $r \in [n]$ για τον οποίο είναι $w(X_{r-1}) < M$ και $w(X_r) \geq M$, οπότε η βέλτιστη λύση είναι η

$$x_1 = \cdots = x_{r-1} = 1, \quad x_r = (M - w(X_{r-1}))/w_r, \quad x_{r+1} = \cdots = x_n = 0.$$

Πράγματι, τότε είναι $w_r x_r = M - w(X_{r-1})$, οπότε $w(X_n) = M$ και η X_n είναι βέλτιστη λόγω της διάταξης $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$. και προφανώς είναι ένα άνω φράγμα για την $optX$.
Κατόπιν τούτων, θέτουμε $B(X_k) = p(X_k) + f(k+1, M - w(X_k))$, όπου ο δεύτερος όρος είναι το κέρδος της άπληστης λύσης για το fractional knapsack problem στο υποσύνολο αντικειμένων $\{k+1, \ldots, n\}$ με βάρος $M - w(X_k)$.

Backtracking - bounding - Maximum clique

Δίδεται (απλό, μη κατευθυνόμενο) γράφημα $G = (V, E)$ και ζητείται να βρεθεί μια μέγιστη κλίκα.
Για κάθε σύνολο $W \subseteq V$, συμβολίζουμε με $G[W]$ το γράφημα που προκύπτει από το G διαγράφοντας όλες τις κορυφές στο $V \setminus W$ καθώς και τις ακμές που έχουν (τουλάχιστον ένα) άκρο στο $V \setminus W$.
Αν $X_k = x_1 x_2 \cdots x_k$ είναι μια μερική λύση, δηλαδή τα στοιχεία της αποτελούν κλίκα, με σύνολο επιλογών C_{k+1}, η οποία επεκτείνεται σε μια κλίκα $X_j = x_1 x_2 \cdots x_k x_{k+1} \cdots x_j$, $j > k$, τότε το σύνολο $\{x_{k+1}, \ldots, x_j\}$ πρέπει να είναι μια κλίκα του $G[C_{k+1}]$ (οι κορυφές αυτές προφανώς ανήκουν στο C_{k+1} και αποτελούν κλίκα, ως υποσύνολο μιας κλίκας). Αν συμβολίσουμε με $mc(k)$ το μέγεθος της μέγιστης κλίκας του $G[C_{k+1}]$, τότε μια συνάρτηση-φράγμα για τη μερική λύση X_k είναι η

$$B(X_k) = k + ub(k),$$

όπου $ub(k)$ ένα άνω φράγμα για την τιμή $mc(k)$.

Backtracking - bounding - Maximum clique

Η εύρεση της τιμής $mc(k)$ είναι τόσο δύσκολη όσο και το αρχικό πρόβλημα, αφού θα πρέπει να λύσουμε το ίδιο πρόβλημα σε ένα μικρότερο γράφημα. Για το λόγο αυτό, χρησιμοποιούμε το άνω φράγμα $ub(k)$, δηλαδή μια υπερεκτίμηση της $mc(k)$.
Απομένει να επιλέξουμε τη μορφή του άνω φράγματος $ub(k)$. Μια προφανής επιλογή είναι να θέσουμε $ub(k) = |C_{k+1}|$ (αφού κάθε κλίκα του $G[C_{k+1}]$ έχει το πολύ $|C_{k+1}|$ στοιχεία), οπότε $B(X_k) = k + |C_{k+1}|$.
Στη συνέχεια, θα δώσουμε ένα καλύτερο (πιο περιοριστικό) άνω φράγμα $ub(k)$, το οποίο βασίζεται στον χρωματισμό του γραφήματος $G[C_{k+1}]$.

Ορισμός

Ένας k-χρωματισμός ενός γραφήματος $G = (V, E)$ είναι μια απεικόνιση $f : V \to [k]$, τέτοια ώστε $\{x, y\} \in E \Rightarrow f(x) \neq f(y)$ (δηλαδή μια ανάθεση ενός χρώματος σε κάθε κορυφή ώστε γειτονικές κορυφές να έχουν διαφορετικό χρώμα).

Αν το γράφημα G έχει έναν k-χρωματισμό, τότε η μέγιστη κλίκα του θα έχει μέγεθος το πολύ k, αφού κάθε κορυφή της πρέπει να έχει διαφορετικό χρώμα από τις υπόλοιπες.
Επομένως, αν χρωματίσουμε το $G[C_{k+1}]$ με λίγα χρώματα, θα έχουμε ένα καλό άνω φράγμα για το $mc(k)$.
Δυστυχώς, το πρόβλημα εύρεσης του ελάχιστου αριθμού k είναι εξίσου δύσκολο (είναι NP-hard) με την εύρεση μέγιστης κλίκας. Όμως, μπορούμε με τον ακόλουθο άπληστο αλγόριθμο να βρούμε γρήγορα έναν k-χρωματισμό που το k θα είναι συχνά αρκετά μικρό.
Ο αλγόριθμος αυτός αναθέτει ένα χρώμα $f(v)$ σε κάθε κορυφή $v \in V$ και επιστρέφει το πλήθος χρωμάτων που χρησιμοποίησε. Πρακτικά, αναθέτει σε κάθε κορυφή το ελάχιστο χρώμα που δεν έχει ήδη ανατεθεί στους γείτονές της.

Backtracking - bounding - Maximum clique

```
1  greedyColor(G = (V, E)) begin
2      k := 1;                                  /* minimum unused color */
3      colorClass(1) := ∅;                      /* no vertex colored 1 yet */
4      foreach v ∈ V do
5          h := 1;
6          while h < k and colorClass(h) ∩ N_v ≠ ∅ do
7              h := h + 1;
8          if h = k then
9              k := k + 1;
10             colorClass(h) := ∅;
11         colorClass(h) := colorClass(h) ∪ {v};
12         f(v) = h;
13     return (k − 1);
```

Αλγόριθμος 10: Άπληστος αλγόριθμος χρωματισμού γραφήματος.

Αν εκτελέσουμε τον αλγόριθμο χρωματισμού στο G, τότε κάθε κορυφή $v \in V$ θα πάρει ένα χρώμα $f(v)$ και θα έχουμε τη συνάρτηση-φράγμα

$$B(X_k) = k + |\{f(v) : v \in C_{k+1}\}|.$$

Εναλλακτικά, μπορούμε να εκτελούμε δυναμικά τον αλγόριθμο χρωματισμού σε κάθε $G[C_{k+1}]$, οπότε θα είναι

$$B(X_k) = k + \mathrm{greedyColor}(G[C_{k+1}]).$$

Η δεύτερη προσέγγιση είναι πιο χρονοβόρα σε χρωματισμούς, αλλά δίνει καλύτερα φράγματα που μπορεί να επιταχύνουν συνολικά την αναζήτηση της μέγιστης κλίκας.

Backtracking - bounding - Maximum clique

```
1  maxClique(k) begin   /* global: X, C_k, A_k, optSize, optClique */
                                          /* X_k = (x_1 x_2, ..., x_k) */
2      if k > optSize then
3          optSize := k;
4          optClique := X_k;
5      if k = 0 then C_{k+1} := V else C_{k+1} := C_k ∩ A_{x_k};
6      M := B(X_k);
7      for y ∈ C_{k+1} do
8          if M ≤ optSize then return;
9          x_{k+1} := y;
10         maxClique(k + 1);
```

Αλγόριθμος 11: Αλγόριθμος backtracking με φράγμα για την εύρεση μέγιστης κλίκας. Αρχική κλήση: maxClique(0).

Backtracking - Ασκήσεις

- Knight's tour: Να υλοποιηθεί αλγόριθμος backtracking για την εύρεση ενός μονοπατιού ενός ίππου σε μια 6×6 σκακιέρα που ξεκινά από την πάνω αριστερή γωνία και επισκέπτεται κάθε τετράγωνο ακριβώς μία φορά.
- Να υλοποιηθεί αλγόριθμος backtracking για την απαρίθμηση των διαφορετικών χρωματισμών των κορυφών ενός γραφήματος με k χρώματα (δύο γειτονικές κορυφές δεν επιτρέπεται να έχουν το ίδιο χρώμα).