Lokální vyhledávání

Abstrakt

Tyto poznámky se zabývají technikou lokálního vyhledávání. Obsahují jak obecný popis techniky, tak i příklady konkrétních algoritmů.

1 Základní princip

Lokální vyhledávání je technika řešení optimalizačních problémů. Možná i proto, že její myšlenka je velmi jednoduchá, je tato technika široce používaná (například pro učící algoritmy neuronové sítě). Algoritmus implementující lokální vyhledávání si po celou dobu běhu udržuje jedno aktuální přípustné řešení. Na začátku si může za aktuální řešení vybrat libovolné z přípustných řešení dané instance (např. náhodně vygenerované). Pak iterativně přechází na další přípustná řešení, které vybírá z okolí toho aktuálního. Pro potřeby algoritmu je nutné přesně určit, jak vypadá okolí libovolného přípustného řešení a také, jak z tohoto okolí jedno řešení vybrat. Iterace končí po splnění vhodné podmínky. Často používané podmínky jsou například: cena aktuálního řešení a řešení, na které přecházíme, se už moc neliší; nemůžeme přejít na řešení s lepší cenou apod. Princip můžeme shrnout v následujícím pseudokódu.

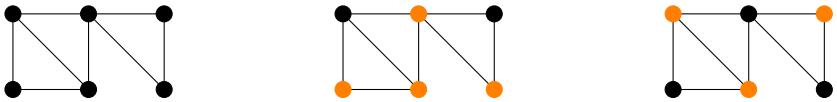
```
1:  procedure LocalSearch(I)
2:      S ← InitialSolution(I)                    ▷ Vygeneruj počáteční řešení
3:      while True do
4:          C ← Neighbourhood(S, I)               ▷ Vygeneruj okolí aktuálního řešení
5:          S' ← SelectNeighbour(C, S, I)         ▷ Najdi další řešení
6:          if End(I, S', S) then                 ▷ Test konce iterace
7:              return S'
8:          end if
9:          S ← S'
10:     end while
11: end procedure
```

Pro konkrétní algoritmus musíme doplnit implementace procedur Neighbourhood, SelectNeighbour a End. Řádky 4 a 5 lze někdy spojit do jednoho, pokud dokážeme nalézt následující řešení bez potřeby celé okolí vygenerovat. Pomocné procedury obecně nejsou vzájemně nezávislé, například způsob volby dalšího řešení může mít vliv na způsob, kterým určujeme konec algoritmu.

Techniku si ukážeme na problému *vrcholového pokrytí grafu* (Vertex Cover). Vstupem je neorientovaný graf. Cílem je najít minimální množinu uzlů takovou, že pokrývá všechny hrany grafu. To znamená, že každá hrana grafu inciduje s některým uzlem z této množiny. Formální definice následuje.

Vrcholové pokrytí grafu	
Instance:	Neorientovaný graf $G = (V, E)$.
Přípustná řešení:	$sol(G) = \{V' \mid V' \subseteq V$ a pro všechny hrany $\{u, v\} \in E$ platí, že buď $u \in V'$ nebo $v \in V'\}$
Cena řešení:	$cost(V', G) = \lvert V' \rvert$
Cíl:	minimum

Jako počáteční pokrytí můžeme zvolit libovolné pokrytí, výpočetně nejjednoduší je zvolit všechny uzly grafu, které tvoří pokrytí vždycky. Jako další je nutné zvolit jak bude vypadat okolí aktuálního řešení. Při volbě je nutné mít na paměti, že v okolí chceme efektivně hledat následující řešení (tedy čím menší okolí, tím lépe). Na druhou stranu musí být okolí dostatečně velké, aby se v něm potenciálně nacházelo dobré řešení. V algoritmu,



(a) Neorientový graf (b) Vrcholové pokrytí grafu (c) Vrcholy netvořící pokrytí

Obrázek 1: Pokrytí grafu

který si ukážeme, za okolí aktuálního pokrytí zvolíme všechna taková pokrytí, která dostaneme přídáním nebo odebráním vrcholu. Pro pokrytí C potom máme

$$\textsc{Neighbourhood}(C,(V,E)) = \{C \setminus \{v\} \mid v \in C,\, C \setminus \{v\} \text{ je pokrytí}\} \cup \{C \cup \{v\} \mid v \in E \setminus C\}. \qquad (1)$$

Pro výběr prvku z okolí existuje několik strategií, z nichž nejjednoduší je vybrat prvek s nejlepší cenou. Této strategii se říká *gradientní metoda*. V případě vrcholového pokrytí tedy vybíráme pokrytí s nejmenším počtem prvků. Protože okolí je definováno jako (1), vybíráme jeden z prvků, které dostaneme odebráním vrcholu z aktuálního pokrytí. Který z nich vybereme je jedno, můžeme vybrat náhodný (pozor, výběr prvku z okolí sice ovlivňuje další běh algoritmu, ale v momentě výběru nemůžeme rozhodnout, který z prvků se stejnou cenou vede k lepšímu finálnímu řešení, takže lze vybrat náhodný prvek). Přirozeně, algoritmus končí v momentě, kdy se v okolí aktuálního řešení nevyskytuje žádné řešení s lepší cenou.

Pseudokód algoritmu následuje:

Algoritmus 1 Vrcholové pokrytí gradientní metodou

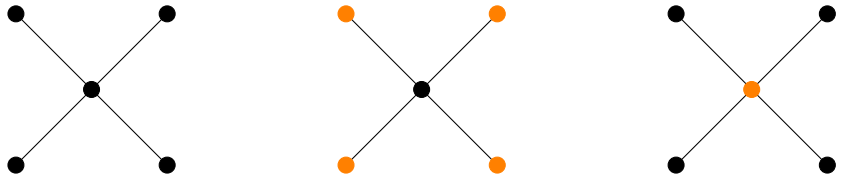
```
1:  procedure VertexCoverGradient((V, E))
2:      C ← V
3:      F ← ∅                                          ▷ Vrcholy, ktere nelze odebrat
4:      while C \ F ≠ ∅ do
5:          Z C \ F vyber náhodný vrchol u
6:          x ←TRUE
7:          for {v, w} ∈ E do                          ▷ Pokrývá C \ {u} všechny hrany?
8:              if v ∉ C \ {u} and w ∉ C \ {u} then
9:                  x ←FALSE
10:                 F ← F ∪ {u}                        ▷ u nelze odebrat
11:                 break
12:             end if
13:         end for
14:         if x then
15:             C ← C \ {u}                            ▷ Přejdu na další řešení
16:         end if
17:     end while
18:     return C
19: end procedure
```

Časová složitost VertexCoverGradient je v nejhorším případě $O(|V| \cdot |E|)$. Cyklus na řádku 4 proběhne maximálně $|V|$ krát, cyklus na řádku 7 maximálně $|E|$ krát.

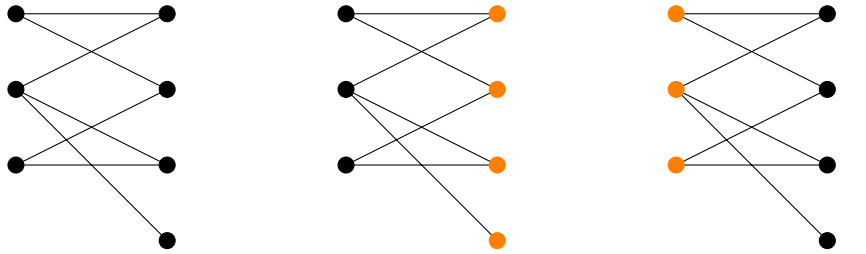
1.1 Problém uváznutí v lokálním minimu

Gradientní metoda nemusí vést k nalezení optimálního řešení. Pokud se algoritmus dostane do situace, kdy se v okolí aktuálního řešení nenachází řešení s lepší cenou, označujeme aktuální řešení jako *lokální minimum*. Optimální řešení pak označujeme jako *globální minimum*. V obecném případě se lokální a globální minimum nemusí shodovat. Může se tedy stát, že algoritmus vrátí řešení, které není optimální. Ukažme si několik příkladů pro VertexCoverGradient.

Prvním příkladem je rodina grafů, které mají tvar hvězdy (star graf). V těchto grafech existuje jeden uzel — střed, který sousedí se všemi ostatními uzly, přičemž ostatní uzly už s žádným jiným uzlem nesousedí. Graf



(a) Star graf (b) Lokální minimum (c) Globální minimum



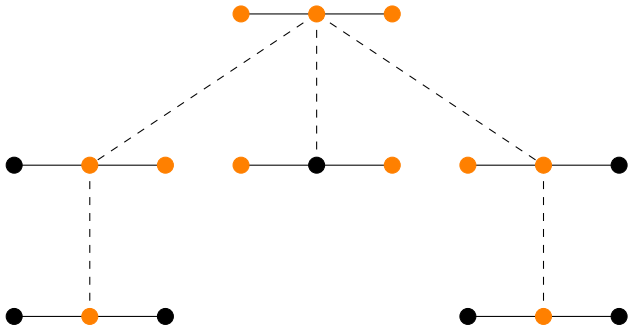
(d) Bipartitní graf (e) Lokální minimum (f) Globální minimum

Obrázek 2: Lokální a globální minima pro vybrané typy grafů

se dá zobrazit tak, že připomíná hvězdu. Pokud VertexCoverGradient vybere v první iteraci jako uzel k odebrání střed, nevyhnutně skončí v lokálním minimu, které není globálním minimem. Pokud vybere v první iteraci libovolný jiný uzel, skončí v globálním minimu.

Dalším příkladem jsou bipartitní grafy. Bipartitní graf je takový graf, jehož uzly můžeme rozdělit do dvou disjuktních množin, přičemž pro každý uzel platí, že může sousedit pouze s uzly z množiny, do které nepatří. Pokud tyto množiny nemají stejný počet uzlů, je globálním minimem menší množina. Lokálních minim může existovat více, příkladem jednoho z nich je větší množina uzlů.

Protože počet iterací cyklu na řádku 4 procedury VertexCoverGradient je konečný (je omezen seshora počtem uzlů v grafu), lze všechny možné průběhy této procedury zakreslit do konečného stromu. Listy tohoto stromu odpovídají lokálním minimům (pozor, více listů může odpovídat stejnému lokálnímu minimu!). Pak platí, že čím hlouběji se list nachází, tím je blíže globálnímu minimu. Listy, které jsou v maximální hloubce odpovídají globálním minimům. Zobrazíme si takový strom pro příklad tříprvkového grafu.



Vidíme, existuje jedno globální minimum (prostřední uzel) a jedno lokální minimum, které není globálním (okrajové uzly).

Techniky, které se používají k částečnému řešení problému uváznutí v lokálním minimu, jsou většinou založeny na fyzikálních analogiích (které si z pochopitelných důvodů nebudeme vysvětlovat). Základní idea je ovšem jednoduchá. Pokud při výběru dalších řešení z okolí aktuálního řešení umožníme zvolit i řešení s horší cenou než aktuální, můžeme někdy zabránit uváznutí tím, že „vyskočíme“ z lokálního minima (nebo z cesty do něj). V dalším průběhu algoritmu se pak lokálnímu minimu, ze kterého jsme vyskočili, můžeme vyhnout. U vrcholového pokrytí by to na přechozím obrázku znamenalo „přeskočit“ do uzlu stromu, který je ve stromu výše (tedy z

Algoritmus 2 Vrcholové pokrytí simulovaným žíháním

```
1: procedure VertexCoverAnnealing((V, E), k)
2:     C ← V
3:     i ← 0
4:     ch ←TRUE                                  ▷ ch je TRUE, pokud jsem změnil v minulé iteraci C
5:     while i ≤ k do
6:         if ch then                                                    ▷ Spocitam okoli C
7:             F ← ∅
8:             for u ∈ C do                          ▷ Odebírání hran, musím testovat na pokrytí
9:                 x ←TRUE
10:                for {v, w} ∈ E do
11:                    if v ∉ C \ {u} and w ∉ C \ {u} then
12:                        x ←FALSE
13:                        break
14:                    end if
15:                end for
16:                if x then
17:                    F ← F ∪ {C \ {u}}
18:                end if
19:            end for
20:            for u ∈ V \ C do                                          ▷ Přidávání hran
21:                F ← F ∪ {C ∪ {u}}
22:            end for
23:        end if
24:        Vyber náhodný prvek S ∈ F                          ▷ s uniformním rozložením
25:        if |S| > |C| then
26:            S pravděpodobností e^{−1/T(i)} proveď C ← S a ch ← TRUE
27:            Jinak ch ← FALSE
28:            continue
29:        end if
30:        i ← i + 1
31:    end while
32:    return C
33: end procedure
```

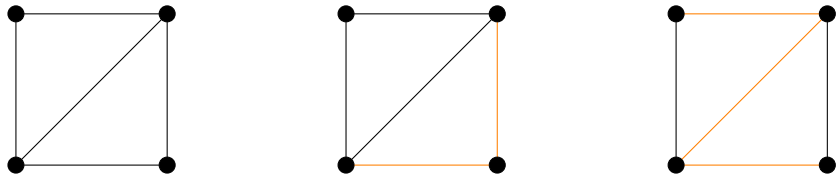
lokálního minima uprostřed bychom se dostali opět do kořene).

Výběr řešení z okolí vypadá následovně. Pro jednoduchost předpokládejme, že řešíme minimalizační problém.

1. Pro aktuální řešení x vybereme náhodné řešení y z okolí x (s co nejvíce uniformním rozložením pravděpodobnosti),
2. pokud $cost(x) < cost(y)$ (y je horší řešení než x), pak s pravděpodobností
$$e^{-(cost(y)-cost(x))/T}, \tag{2}$$
kde T je konstanta, vybereme jako další řešení y (alternativou je ponechání x),
3. pokud je $cost(x) \geq cost(y)$, je dalším řešením je y.

Konstanta T určuje míru nestability výběru, čím je T vyšší, tím vyšší je i (2). Všimněte si také, že čím menší je rozdíl mezi cenami x a y, tím je pravděpodobnost výběru y vyšší. Vysvětlením může být to, že pokud se blížíme k lokálnímu minimu, většinou je rozdíl v cenách aktuálního řešení a prvků z jeho okolí menší než pokud jsme dále od něj. Existuje tedy větší šance, že se blížíme k uváznutí v lokálním minimu. Popsaný přístup označujeme jako *Metropolis algoritmus*. Vzhledem ke změně výběru dalšího řešení musíme také změnit podmínku ukončení algoritmu. Algoritmus nemůžeme ukončit stejně jako u gradientní metody (můžeme se totiž nacházet v lokálním minimu a ukončením algoritmu bychom zabránili možnosti z něj vyskočit). Mezi často používané podmínky ukončení algoritmu patří zejména:

- algoritmus provede předem daný počet iterací,



(a) Ohodnocený graf (b) Řez s cenou 2 (c) Řez s cenou 3

Obrázek 3: Řez grafu

- algoritmus dostatečný počet iterací za sebou nevybere jiné řešení než aktuální.

Metropolis algoritmu obecně trvá poměrně dlouhou dobu, než se ustálí (tj. trvá poměrně hodně iterací, než se dostane k lokálnímu (globálnímu) minimu). Důvodem je fakt, že i když se nachází blízko minima, pravděpodobnost (2) je poměrně vysoká. Tento problém se dá vyřešit tak, že s rostoucím počtem iterací bude (2) klesat (pochopitelně pro stejné x a y). V počátečních iteracích tak exituje poměrně velká šance, že algoritmus unikne z lokálního minima se špatnou cenou, ale v pozdějších iteracích už bude (2) dostatečná malá pro to, aby se algoritmus zastavil v minimu s lepší cenou. Technicky to lze zajistit tak, že konstantu T nahradíme klesající funkcí závislou na počtu iterací. Popsaná úprava má jméno *simulované žíhání.*

Na závěr kapitoly si ukážeme úpravu VertexCoverGradient na techniku simulovaného žíhání (Algoritmus 2). Abychom dosáhli uniformního rozložení pravděpodobnosti při volbě na řádku 24, vygeneruje algoritmus na řádcích 6 až 23 okolí aktuálního řešení C. Toto okolí se generuje jenom v případě, že se v předchozí iteraci C změnilo (proměnná *ch*). Za $T(i)$ je možné zvolit libovolnou klesající funkci takovou, aby $0 \leq e^{-1/T(i)} \leq 1$. K nalezení vhodné funkce je v mnoha případech nutné experimentování.

2 Příklady algoritmů

2.1 Maximální řez

Problém nalezení maximálního řezu v grafu je notoricky známým a dobře prostudovaným problémem, který lze snadno řešit pomocí lokálního vyhledávání. Zajímavé je, že pro výběr následujícího prvku z okolí aktuálního řešení se nepoužívá gradientní metoda ani simulované žíhání. Tato vylepšení nemají totiž na výkon algoritmu (tj. jak dobrý řez vrací) nijak zásadní vliv.

Definice 1. Nechť $G = (V, E)$ je neorientovaný graf a V_1, V_2 jsou disjunktní podmnožiny V takové, že $V_1 \cup V_2 = V$. Pak množinu hran $C = \{(u, v) \mid u \in V_1, v \in V_2\}$ označujeme jako *řez grafu*. Cenou řezu C je $|C|$.

Příklady pojmu z předchozí definice jsou na obrázku 3. Problém nalezení maximálního řezu v grafu je definován následovně

Maximální řez grafu	
Instance:	Neorientovaný graf $G = (V, E)$.
Přípustná řešení:	$\{C \subseteq E \mid C$ je řez grafu $G\}$
Cena řešení:	$cost(C, G) = \|C\|$
Cíl:	maximum

Aproximační algoritmus, který využívá techniku lokálního vyhledávání, si v každém kroku uchovává množiny V_1 a V_2 a jim odpovídající řez C. Na začátku algoritmus zvolí tyto množiny V_1 a V_2 libovolně, například $V_1 = V$, $V_2 = \emptyset$. Okolí aktuálního řezu C jsou všechny řezy, které vzniknou přesunem jednoho uzlu takového, že počet hran, které vedou z uzlu do množiny, ve které se nachází, je větší, než počet hran, které vedou do opačné množiny, mezi množinami V_1 a V_2. Z okolí pak algoritmus vybírá libovolný řez.

V pseudokódu používáme následující značení. Pro uzel v je V_v ta z množin V_1, V_2, do které tento prvek patří.

Věta 1. *Složitost algoritmu* MaxCut *je* $O(|V| \cdot |E|)$.

Důkaz. Cyklus na řádku 7 se dá provést se složitostí $O(|V|)$. Protože maximální cena řezu je $|E|$ a v každé iteraci cyklu na řádku 5 zvýšíme cenu řezu minimálně o 1, je maximální počet iterací tohoto cyklu $|E|$. □

Algoritmus 3 Maximální řez pomocí lokálního vyhledávání

```
1: procedure MaxCut((V, E))
2:     V_1 ← V
3:     V_2 ← ∅
4:     x ←TRUE
5:     while x do
6:         x ←FALSE
7:         for u ∈ V do
8:             if |{{u, v} | V_v = V_u}| > |{{u, v} | V_u ≠ V_u}| then
9:                 Přesun u mezi V_1 a V_2
10:                x ←TRUE
11:                break
12:            end if
13:        end for
14:    end while
15:    return C odpovídající V_1 a V_2.
16: end procedure
```

2.2 Uniformní rozdělení grafu

Definice 2. Nechť $G = (V, E)$ je neorientovaný jednoduchý graf, kde $|V| = 2n$ pro přirozené n. Dále uvažujme ohodnocovací funkci $c : E \to \mathbb{Q}$. Uniformní rozdělení grafu je tvořeno množinami vrcholů A a B takovými, že $A \cup B = V$ a současně $|A| = |B| = n$. Problém uniformního rozdělení grafu je problémem nalezení uniformního rozdělení, které minimalizuje

$$cost(A, B) = \sum_{u \in A v \in B} c(\{u, v\}).$$

Nechť A^*, B^* tvoří optimální uniformní rozklad grafu a A, B tvoří rozklad, který není optimální. Označme $Y = B^* \cap A$ (Y je tvořeno prvky, které v optimálním rozdělení patří do B^* ale v rozdělení tvořeném A, B patří do A), a $X = A^* \cap B$ (X je tvořeno prvky, které v optimálním rozdělení patří do A^* ale v rozdělení tvořeném A, B patří do B). Všimněme si, že platí $|X| = |Y|$ a $A^* = (A - X) \cup Y$, $B^* = (B - Y) \cup X$. Od rozdělení A, B můžeme tedy přejít k optimálnímu rozdělení pomocí prohození X a Y.

Definice 3. Nechť A, B je uniformní rozdělení, a $a \in A$, $b \in B$. Pokud na A, B aplikujeme operaci prohození a a b, dostaneme rozdělení $A' = (A - \{a\}) \cup b$, $B' = (B - \{b\}) \cup a$.

Pro operaci prohození a a b můžeme určit změnu, kterou takové prohození bude mít na cenu rozdělení. Pro uzel $a \in A$ definujeme

$$E(a) = \sum_{u \in B} c(\{a, u\})$$
$$I(a) = \sum_{u \in A} c(\{a, u\})$$
$$D(a) = E(a) - I(a)$$

Číslo $E(a)$ je sumou cen hran incidujících s a, které v rozdělení A, B vedou do množiny B. Tyto hrany přispívají k ceně rozdělení A, B, k ceně rozdělení A', B' ovšem nepřispívají. Na druhou stranu, $I(a)$ je sumou cen hran incidujících s a, které v rozdělení A, B zůstavají v podgrafu generovaném vrcholy A. Tyto hrany nepřispívají k ceně rozdělení A, B, přispívají ovšem k ceně rozdělení A', B'. Hodnota $D(a)$ je poté změna ceny způsobená přesunem prvku a z A do B, tedy rozdílu $cost(A, B) - cost(A', B')$.

Lemma 1. *Prohození prvků a, b vede ke změně ceny rozkladu o*

$$g(a, b) = D(a) + D(b) - 2c(\{a, b\})$$

Důkaz. Přesuneme a z A do B.

Hodnota $I'(a)$ vzhledem k rozdělení $(A-\{a\}, B\cup\{a\})$ je rovna $E(a)$ vzhledem k (A, B). Naopak hodnota $E'(a)$ vzhledem k rozdělení $(A-\{a\}, B\cup\{a\})$ je rovna $I(a)$ vzhledem k (A, B). Cena rozdělení se tedy změní o $D(a)$.

Pro prvek b nyní (v rozdělění $(A-\{a\}, B\cup\{a\})$) máme $E'(b) = E(b) - c(\{a, b\})$ a $I'(b) = I(b) - c(\{a, b\})$. Odtud dostáváme, že cena přesunu b je $D'(b) = E'(b) - I'(b) = D(b) - 2c(\{a, b\})$.

Změna ceny prohození a a b se rovná sumě změny ceny způsobené přesunem a a ceny změny způsobené přesunem b, tedy je rovna $D(a) + D(b) - 2c(\{a, b\})$. □

Operaci prohození můžeme využít ke generování okolí, které použijeme pro algoritmus využívající lokální vyhledávání. Okolí rozdělení A, B je tvořeno všemi rozděleními, které můžeme obdržet pomocí jedné operace prohození. Poté stačí použít gradientní metody a obdržíme algoritmus s kvadratickou složitostí. Z experimentální zkušenosti víme, že pro grafy s více než 32 uzly a cenou hran $c : E \to \{0, 1\}$ se v 10 procentech případů dostaneme k optimálnímu rozdělení, v 75 procentech případů je řešení o 1 nebo 2 horší. Ukážeme si, způsob, jakým úspěšnost nalezení optimálního řešení vyplepšit.

Úspěšnost nalezení optimálniho řešení můžeme zvýšit zvětšením okolí, tj. do okolí dáme i rozdělení, která obdržíme pomocí více prohození. Bohužel tím zvýšíme i složitost algoritmu, už pro okolí generované maximálně 2 prohozeními je složitost $O(n^4)$. Tento problém vyřešíme tak, že budeme generovat sekvence prohození pomocí greedy pravidla.

Algoritmus pro nalezení sekvence prohození (vstupem je rozdělení A, B):

1. Spočítej $D(v)$ pro všechny prvky $v \in V$
2. Vyber pár a_i, b_i (i udává počet provedených iterací) takový, že změna

   $$g_i = D(a_i) + D(b_i) - 2c(\{a_i, b_i\})$$

   je maximální.
3. prohoď a_i a b_i a přepočítej $D(v)$:

   $$D'(x) = D(x) + 2c(\{x, a_i\}) - 2c(\{x, b_i\}) \text{ pro } x \in A - \{a_i\}$$
   $$D'(y) = D(y) + 2c(\{y, b_i\}) - 2c(\{y, a_i\}) \text{ pro } y \in B - \{b_i\}$$

4. Opakuj kroky 2 a 3 (jednou použité prohození už znovu v kroku 2 neuvažujeme), a obdržíš sekvenci prohození $a_1, b_1; a_2, b_2; \ldots a_n, b_n$.
5. Označme $G(k) = \sum_{j=1}^{k} g_i$. Vrať sekvenci prohození, $a_1, b_1; a_2, b_2; \ldots a_k, b_k$, pro kterou je $G(k)$ maximální.

Algoritmus nalezení uniformního rozdělení probíhá následovně:

1. Vyber náhodné rozdělení A, B
2. Pomocí předchozího algoritmu spočítej sekvenci prohození s maximálním $G(k)$
3. Pokud je $G(k) > 0$ prohození proveď a opakuj krok 2, v opačném případě algoritmus končí.

Reference

[1] Donald Knuth. The Art of Computer Programming, Volume I. Addison-Wesley, 1997

[2] Donald Knuth. Selected papers on analysis of algorithms. Center for the study of language an information, 2000

[3] Cormen et. al. Introduction to algorithms. The MIT press. 2008.

[4] Donald Knuth Concrete mathematics: A foundation for computer science. Addison-Wesley professional, 1994

[5] John Kleinberg, Éva Tardes. Algorithm design. Addison-Wesley, 2005.

[6] U. Vazirani et. al. Algorithms. McGraw-Hill, 2006.

[7] Steve Skiena. The algorithm design manual. Springer, 2008.

[8] Juraj Hromkovič. Algorithmics for hard problems. Springer, 2010.

[9] Oded Goldreich. Computational complexity: a conceptual perspective. Cambridge university press, 2008.