Metoda hrubé síly, backtracking a branch-and-bound

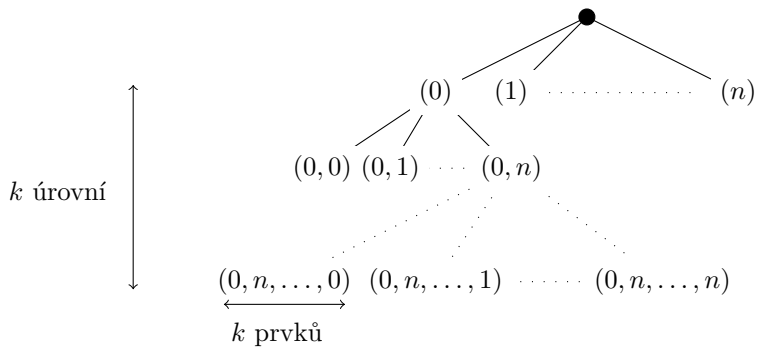
Abstrakt

Tyto poznámky se zabývají technikami použití hrubé síly. Obsahují jak obecný popis techniky, tak i příklady konkrétních algoritmů.

1 Základní princip

Metoda hrubé síly se dá popsat jednoduchou větou „zkus všechny možnosti." U optimalizačních problémů to znamená, že k dané vstupní instanci algoritmus vygeneruje všechna přípustná řešení a pak z nich vybere to optimální. Například u úlohy batohu, kde chceme najít podmnožinu vah takovou, že její suma je maximální ze všech podmnožin, jejíchž suma je menší než kapacita, algoritmus používající hrubou sílu nejdříve vygeneruje všechny podmnožiny vah a pak najde tu nejlepší. Technika hrubé síly je použitelná i pro rozhodovací problémy. Pokud pro danou instanci x rozhodovacího problému L (zde chápaného jako jazyk L), existuje vhodný certifikát, na jehož základě víme, že $x \in L$ (tedy, odpověd je ano), algoritmus fungující hrubou silou vygeneruje všechny možné kandidáty na takový certifikát a poté se jej v této množině pokusí najít. Uvažme například problém SAT. Zde můžeme za certifikát považovat takové ohodnocení výrokových proměnných, při kterém je vstupní instance, tj. formule výrokové logiky, pravdivá. Algoritmus vygeneruje všechna možná ohodnocení výrokových proměnných této formule a poté ověří, jestli je pro některé z nich formule pravdivá. Pokud takové ohodnocení najde, je odpověď *ano*.

Generování všech možností vede ve většině případů ke generování základních kombinatorických struktur jako jsou permutace a variace. Způsob jejich generování si můžeme ukázat na klasickém problému tahání barevných míčků z pytlíku. Předpokládejme, že máme míčky n barev označených jako $\{0, 1, \ldots, n-1\}$ a chceme vytáhnout k míčků. Algoritmus pro vygenerování všech možností vytažení těchto míčků prochází následující strom.



Uzly stromu odpovídají vytažení jednoho míčku. Podle úrovní stromu určíme, kolikátý míček v pořadí taháme. Označíme-li si generovanou sekvenci jako $\langle a_1, \ldots, a_k \rangle$, pak 1. úroveň (uzly v hloubce 1) odpovídá volbě a_1, 2. úroveň odpovídá volbě pro a_2 a tak dále. Pro každý uzel platí, že uzly cestě od kořene k tomuto uzlu jednoznačně určují odpovídající část generované sekvence. Například u uzlu v hloubce 3 víme, které míčky jsme vytáhli pro a_1, a_2 a a_3. Na základě této informace můžeme rozhodnout, jaké míčky zvolíme v další vrstvě (jestli chceme, aby se míček v sekvenci opakoval, nebo ne). Uzly v hloubce k už nemají potomky (a jsou tedy listy). Sekvence odpovídající cestám z kořene do listů jsou pak vygenerované sekvence.

Algoritmus pro generování prochází strom do hloubky, můžeme proto použít rekurzi. Argumenty procedury Generate jsou množina míčků X, doposud sestavená část sekvence $\langle a_1, \ldots, a_i \rangle$ a počet míčků v úplné sekvenci k. Procedura Filter slouží k výběru toho, které míčky lze dosadit za a_{i+1}. Pokud Filter vždy vrátí X, Generate generuje k-prvkové variace s opakováním, pokud Filter vrátí $X - \{a_1, \ldots, a_i\}$, Generate generuje variace bez opakování. Generování spustíme zavoláním Generate s prázdnou sekvencí a.

Pojmenování *backtracking* je inspirováno principem, na kterém Generate pracuje. Když algoritmus nalezne jednu sekvenci (dostane se do listu stromu), vystoupí z rekurze o úroveň nahoru (tj. posune se ve stromě po

Algoritmus 1 Generování variací a permutací

```
1: procedure GENERATE(X, ⟨a_1, ..., a_i⟩, k)
2:     if i = k then                          ▷ Sekvence má k prvků, skonči rekurzi
3:         Zpracuj a_1, ..., a_i
4:     end if
5:     S ←FILTER(X, ⟨a_1, ..., a_i⟩)          ▷ Vyfiltruj prvky, které lze dosadit za a_{i+1}
6:     for x ∈ S do
7:         GENERATE(X, ⟨a_1, ..., a_i, x⟩, k)  ▷ Doplň sekvenci o x a pokračuj v rekurzi
8:     end for
9: end procedure
```

cestě směrem ke kořenu) a generuje další sekvence rekurzivním voláním na řádku 7. Tento „návrat nahoru“ má anglické jméno backtrack.

Pokud používáme backtracking pro řešení konkrétního problému, často nemusíme generovat všechny možnosti. Předpokládejme že máme vygenerovánu část sekvence $a = a_1, \ldots, a_i$. Potom můžeme GENERATE obohatit o následující testy.

I. test na řádku 2 můžeme nahradit testem, který rozhodne, zda-li je $a_1, \ldots, a_i$ už řešením, nebo se dá rychle doplnit na řešení (rychle většinou znamená libovolným výběrem zbylých prvků sekvence).

II. před rekurzivním voláním na řádku 7. můžeme testovat, jestli $a_1, \ldots, a_i, x$ je prefixem sekvence, kterou chceme vygenerovat (tj. to, jestli má smysl pokračovat v generování zbytku sekvence). Pokud ne, GENERATE už rekurzivně nevoláme.

Oba předchozí testy se dají k ořezání stromu rekurzivního volání. Jako ukázku si uvedeme algoritmus pro problém SAT. Připomeňme, že SAT je definován jako

SAT	
Instance:	formule výrokové logiky v konjunktivní normální formě φ
Řešení:	1 pokud je φ splnitelná, jinak 0.

K sestavení algoritmu pro SAT stačí upravit GENERATE. Algoritmus totiž generuje postupně všechna možná ohodnocení výrokových proměnných. Uvažme formuli φ, která je konjunkcí m literálů $C_1, \ldots, C_m$ a obsahuje k výrokových proměnných $x_1, \ldots, x_k$. Ohodnocení těchto proměnných můžeme chápat jako sekvenci $\langle a_1, \ldots, a_k \rangle$ složenou z 1 a 0, přitom a_i je ohodnocení proměnné x_i. Dále si pro klauzuli C_j definujeme následující dva predikáty

- $\mathcal{F}(C_j, \langle a_1, \ldots, a_i \rangle)$ je pravdivý, právě když proměnné obsažené v C_j patří do $\{x_1, \ldots, x_i\}$ a vzhledem k ohodnocení $\langle a_1, \ldots, a_i \rangle$ neobsahuje C_j žádný pravdivý literál,
- $\mathcal{T}(C_j, \langle a_1, \ldots, a_i \rangle)$ je pravdivý, pokud literál alespoň jedné proměnné z C_j patřící do $\{x_1, \ldots, x_i\}$ je při ohodnocení $\langle a_1, \ldots, a_i \rangle$ pravdivý.

S pomocí $\mathcal{F}$ a $\mathcal{T}$ můžeme pro SAT jednoduše implementovat testy zmíněné v bodech I a II. Víme, že formule φ je pravdivá, právě když jsou pravdivé všechny klausule, což platí, právě když je v každé klausuli alespoň jeden pravdivý literál. Pokud je tedy $\mathcal{T}(C_j, \langle a_1, \ldots, a_i \rangle)$ pravdivá pro všechny klausule, víme, že φ je pravdivá pro libovolné doplnění $\langle a_1, \ldots, a_i \rangle$. Naopak, pokud je splněno $\mathcal{F}(C_j, \langle a_1, \ldots, a_i \rangle)$ alespoň pro jednu klausuli, je tato klausule pro libovolné doplnění $\langle a_1, \ldots, a_i \rangle$ nepravdivá, v důsledku čehož je nepravdivá i φ.

Časová složitost EASYSAT je v nejhorším případě exponenciální vhledem k počtu proměnných k. Pokud každá klausule nesplnitelné formule φ obsahuje literál proměnné x_k, pak EASYSAT vygeneruje všechna ohodnocení. Těchto ohodnocení je 2^k.

SAT je důležitý nejen teoreticky (jak zjistíte v kurzu složitosti), ale také pro praktické nasazení. Mnoho v praxi důležitých úloh přechází na SAT (ověřování návrhu logických obvodů, ověřování správnosti modelů apod). Existuje dokonce každoroční soutěž programů, tzv. SAT solverů, pro řešení SAT problému. Mnohé z nich jsou

Algoritmus 2 Jednoduchý SAT solver

```
1: procedure EasySat(φ = C_1 ∨ ··· ∨ C_m, ⟨a_1, ..., a_i⟩, k)
2:     E ←TRUE
3:     for j ← 1 to m do
4:         if not T(C_j, ⟨a_1, ..., a_i⟩) then
5:             E ←FALSE
6:             break
7:         end if
8:     end for
9:     if E then
10:        return TRUE                          ▷ Všechny klauzule jsou pravdivé
11:    end if
12:    for x ∈ {0, 1} do
13:        E ←TRUE                              ▷ Hledám nesplnitelnou klausuli
14:        for j ← 1 to m do
15:            if F(C_j, ⟨a_1, ..., a_i, x⟩) then
16:                E ←FALSE
17:                break
18:            end if
19:        end for
20:        if E then
21:            if EasySat(φ, ⟨a_1, ..., a_i, x⟩, k) then
22:                return TRUE                  ▷ φ je pravdivá, algoritmus končí
23:            end if
24:        end if
25:    end for
26:    return FALSE                             ▷ Obě rekurzivní volání vrátila FALSE
27: end procedure
```

vyvíjené velkými firmami a licencovány za nemalý peníz. EasySat je z tohoto pohledu velmi naivním algoritmem (i když většina SAT solverů je více či méně založena na backtrackingu). Jednoduše ale demonstruje základní princip ořezání stromu rekurze.

2 Příklady algoritmů

2.1 Úloha n dam

Úloha n dan je problém, který se často vyskytuje v programovacích nebo matematických soutěžích. Úkolem je spočítat kolika způsoby lze umístit n dam na šachovnici tak, aby se vzájemně neohrožovaly. Přitom předpokládáme, že dáma může táhnout jako v šachu, tedy posunout se o libovolný počet polí vodorovně, svisle nebo diagonálně.

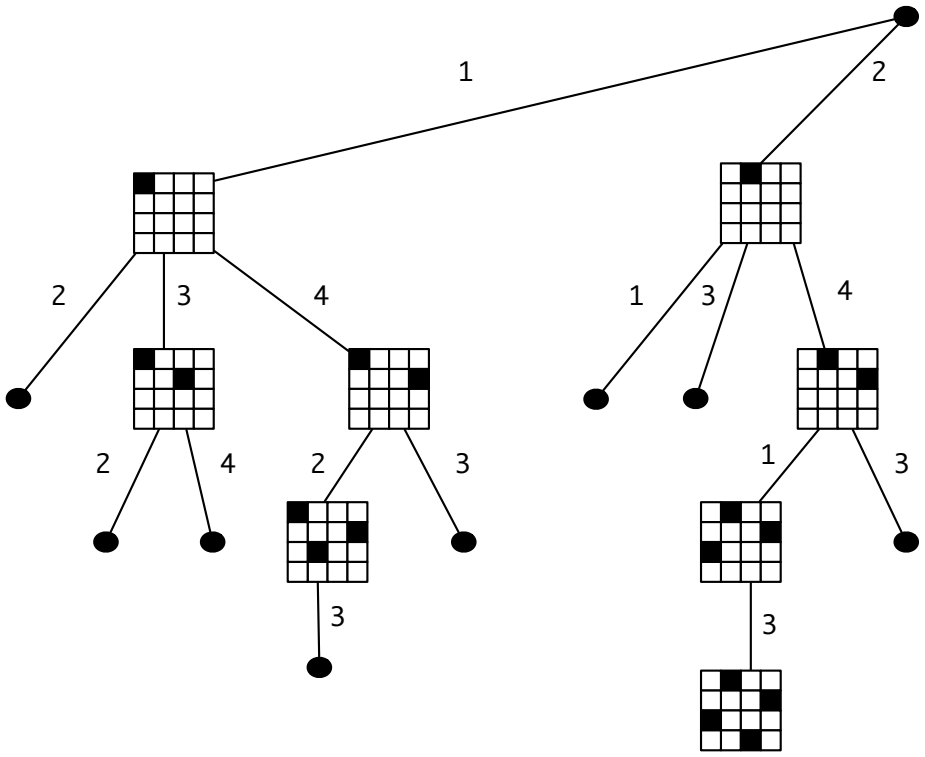
Naivní přístup k řešení tohoto problému by bylo vygenerovat všechna možná rozmístění dam na šachovnici a poté pro každé rozmístění otestovat, jestli je dámy neohrožují. Tento přístup je ale velmi neefektivní, protože takových rozmístění je $\binom{n^2}{n} = \frac{n^2!}{n!(n^2-n)!}$.

Ukážeme si sofistikovanější přístup, ve kterém využijeme znalostí šachových pravidel už během generování. Sloupce a řádky šachovnice si označíme čísly $1 \dots n$. Protože dáma táhne horizontálně, ve správném rozestavení musí být na každém řádku právě jedna dáma. Pozice tedy můžeme generovat jako sekvence $\langle a_1, \dots, a_n \rangle$, kde a_i je sloupec, ve kterém se nachází dáma na i-tém řádku. Protože v každém sloupci může být právě jedna dáma, můžeme jak kostru algoritmu využít Generate pro generování permutací. Jediná věc, kterou můžeme přidat, je test odpovídají podmínce II. Pro $a_1, \dots, a_i$ otestujeme, jestli jestli se dámy v prvních i řádcích neohrožují diagonálně.

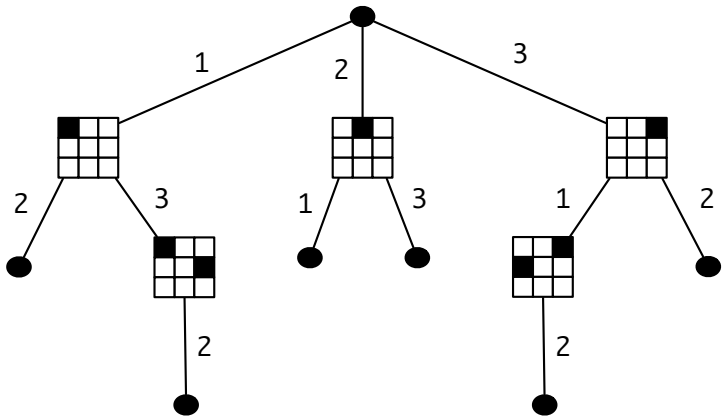
Příklad 1. (a) pro $n = 4$ existují 2 pozice, strom na následujícím obrázku je symetrický

Algoritmus 3 Problém n královen

```
1: procedure QUEENS(⟨a_1, ..., a_i⟩, n)
2:     if i=n then
3:         return 1
4:     end if
5:     S ← {1, ..., n} \ {a_1, ..., a_i}
6:     c ← 0
7:     for a_{i+1} ∈ S do
8:         p ←TRUE
9:         for j ← 1 to i do
10:            if |j − (i + 1)| = |a_j − a_{i+1}| then
11:                p ←FALSE
12:                break
13:            end if
14:        end for
15:        if p then
16:            c ← c + QUEENS(⟨a_1, ..., a_i, a_{i+1}⟩, n)
17:        end if
18:    end for
19:    return c
20: end procedure
```



(a) pro $n = 3$ je výsledek 0.



Složitost Queens není známa, nicméně je obrovská. Jenom počet možných správných pozic a tedy i vygenerovaných úplných sekvencí roste obrovsky rychle. Přesný vzorec není zatím znám, nicméně pro $n = 5$ je správných pozic 10, pro $n = 24$ je jich už více než $227 \cdot 10^{12}$.

2.2 Set Cover

Na problému *Set Cover* si ukážeme jednoduchou techniku, jak prořezat strom rekurze při řešení optimalizačního problému. Řekněmě, že hledáme řešení instance I. Idea je, že budeme generovat prvky $x \in sol(I)$, počítat pro ně cenu $cost(x, I)$ a vybereme ten optimální. Pokud si v průběhu algoritmu pamatujeme cenu zatím nejlepšího nalezeného řešení, lze za určitých podmínek prořezat strom rekurze a vyhnout se výpočtu některých prvků $sol(I)$. Nutnou podmínkou je monotonie funkce *cost* vzhledem k tomu, jak rozšiřujeme hledanou sekvenci. Předpokládejme tedy, že *cost* je neklesající. To znamená, že vždycky platí

$$cost(\langle a_1, \ldots, a_i \rangle, I) \leq cost(\langle a_1, \ldots, a_i, a_{i+1} \rangle, I).$$

Pak, pokud je I instance minimalizačního problému a $cost(\langle a_1, \ldots, a_i \rangle, I)$ je větší než cena doposud nejlepšího nalezeného řešení, tak $\langle a_1, \ldots, a_i \rangle$ není možné doplnit na optimální řešení (cena optimální řešení je menší nebo rovna ceně nejlepšího doposud nalezeného řešení). Duální princip platí pro nerostoucí cenovou funkci a maximalizační problémy.

Pokud existuje efektivní aproximační algoritmus k danému optimalizačnímu problému, je cenu řešení, které tento algoritmus vrátí, možné použít k inicializaci ceny doposud nejlepšího nalezeného řešení. Přístupy popsané v předchozích dvou odstavcích se v literatuře označují jako *branch-and-bound*.

Set cover je „slavný“ optimalizační problém, hojně studovaný především v oblasti aproximačních algoritmů. Jeho zadání je jednoduché. Jsou dány množina X a systém jejích podmnožin $\mathcal{S}$, který tuto množinu pokrývá (tj. platí $\bigcup \mathcal{S} = X$). Úkolem je nalézt co nejmenší podmnožinu $\mathcal{S}$ tak, aby stále pokrývala X.

Set Cover	
Instance:	konečná množina X, systém podmnožin $\mathcal{S} = \{S_i \mid S_i \subseteq X\}$ takový, že $\bigcup_{S_i \in \mathcal{S}} S_i = X$
Přípustná řešení:	$\mathcal{C} \subseteq \mathcal{S}$ takové, že $\bigcup_{S_i \in \mathcal{C}} S_i = X$
Cena řešení:	$cost(X, \mathcal{S}, \mathcal{C}) = \lvert\mathcal{C}\rvert$
Cíl:	minimum

Idea algoritmu je tedy generovat podmnožiny $\mathcal{S}$, které pokrývají X a vybrat tu s nejmenším počtem prvků. K tomuto účelu můžeme opět použít jako kostru Generate. Podmnožiny n prvkové množiny totiž jednoznačně odpovídají n prvkovým variacím nad $\{0, 1\}$. Uvažme množinu Y. Pak pro každou podmnožinu Z této množiny

můžeme definovat její *charakteristickou funkci* $\mu_Z : Y \to \{0,1\}$ jako

$$\mu_Z(y) = \begin{cases} 0 & y \notin Z \\ 1 & y \in Z \end{cases}$$

Mezi charakteristickými funkcemi a podmnožinami je jednoznačný vztah. Každá podmnožina Z indukuje unikátní charakteristickou funkci, a naopak, je-li dána charakteristická funkce $\mu : Y \to \{0,1\}$ pak množinu $Set(\mu)$, která tuto funkci indukuje nalezneme jako

$$Set(\mu) = \{y \in Y \mid \mu(y) = 1\}.$$

Pokud zafixujeme pořadí prvků v množině X, tj $X = \{x_1, x_2, \ldots, x_n\}$, dá se každá podmnožina $Z \subseteq X$ jednoznačně zapsat jako sekvence $\langle \mu_Z(x_1), \mu_Z(x_2), \ldots, \mu_Z(x_n) \rangle$. Ke vygenerování všech podmnožin $\mathcal{S}$ tedy stačí generovat všechny n-prvkové sekvence nad $\{0,1\}$. V dalším budeme značit jako $Set(\langle a_1, \ldots, a_i \rangle)$ podmnožinu, která odpovídá sekvenci $\langle a_1, \ldots, a_i \rangle$ doplněné na konci potřebným počtem 0.

Zamysleme se nad podmínkami, pomocí nichž lze ořezat strom rekurze. Uvažujme situaci, kdy máme vygenerovánu sekvenci $\langle a_1, \ldots, a_i \rangle$. Potom můžeme využít následující (nezapomínejme, že pracujeme s množinami, jejichž prvky jsou podmnožiny X, takže $Set(\langle a_1, \ldots, a_i \rangle)$ je podmnožina $\mathcal{S}$).

- Pokud $\bigcup Set(\langle a_1, \ldots, a_i \rangle) \cup \bigcup(\mathcal{S} \setminus \{S_1, \ldots, S_i\}) \neq X$, pak můžeme rekurzi ukončit, protože $\langle a_1, \ldots, a_i \rangle$ není možné doplnit tak, aby pokryla celou množinu X.
- Pokud $\bigcup Set(\langle a_1, \ldots, a_i \rangle) = X$, tak končíme rekurzi, protože hledáme minimální pokrytí X a přidávat další prvky do $Set(\langle a_1, \ldots, a_i \rangle)$ proto nemá smysl.
- Pokud je $|Set(\langle a_1, \ldots, a_i \rangle)|$ větší než velikost doposud nejmenšího nalezeného pokrytí, končíme rekurzi. Hledáme minimální pokrytí a pokračovat v přidávání prvků do $Set(\langle a_1, \ldots, a_i \rangle)$ nemá smysl.

Algoritmus 4 Set Cover pomocí branch-and-bound

```
1:  C ← S                                  ▷ Globální proměnná pro uchování zatím nejlepšího nalezeného pokrytí
2:
3:  procedure OptimalSetCover(⟨a_1, ..., a_i⟩, F, X)
4:      Y ← ⋃ Set(⟨a_1, ..., a_i⟩)
5:      if Y = X then                      ▷ test, jestli ⟨a_1, ..., a_i⟩ už neindukuje pokrytí
6:          C ← Set(⟨a_1, ..., a_i⟩)
7:          return                         ▷ ⟨a_1, ..., a_i⟩ je zatím nejlepší pokrytí
8:      end if
9:      if Y ∪ ⋃F ≠ X or |Set(⟨a_1, ..., a_i⟩)| = |C| − 1 then
10:         return                         ▷ nemůžu vytvořit lepší pokrytí než je C
11:     end if
12:     OptimalSetCover (⟨a_1, ..., a_i, 1⟩, F \ {S_{i+1}}, X)
13:     OptimalSetCover (⟨a_1, ..., a_i, 0⟩, F \ {S_{i+1}}, X)
14: end procedure
```

Výpočet optimálního řešení spustíme pomocí OptimalSetCover$(\langle\rangle, \mathcal{S}, X)$. Algoritmus si uchovává v globální proměnné $\mathcal{C}$ zatím nejlepší nalezené pokrytí, které na začátku nastaví na celou množinu $\mathcal{S}$ Na řádku 5 pak testujeme, jestli jsme nalezli pokrytí. Pokud ano, je toto pokrytí určitě lepší než $\mathcal{C}$. Na řádku 9 totiž testujeme velikost doposud vygenerované podmnožiny. Pokud je jenom o 1 menší než $\mathcal{C}$, tak ukončíme rekurzi. Přidáním dalšího prvku bychom totiž mohli dostat jenom pokrytí, které je alespoň tak velké jako $\mathcal{C}$. Na stejném řádku také testujeme, zda-li lze aktuální podmnožinu rozšířit na pokrytí. Pokud ne, rekurzi ukončíme.

2.3 Úloha batohu

Použití backtrackingu vede většinou (u všech příkladů, které jsme si ukázali, tomu tak bylo) k algoritmu s horší než polynomickou složitostí. Toto je přirozené, backtracking většinou používáme k nalezení (optimálního) řešení problému, které považujeme za prakticky neřešitelný (jako SAT, Set Cover nebo úloha n dam). Mohlo by se tedy zdát, že bactracking není pro větší instance příliš použitelný. V této kapitole si ale ukážeme, jak

se dá backtracking zkombinovat s algoritmem navrženým jiným způsobem a vylepšit tak jeho výkon (tak, že algoritmus spočítá řešení, které je v průměrném případě bližší optimálnímu řešení). Ukážeme si to na příkladu úlohy batohu. Tento optimalizační problém definovaný jako

Úloha batohu	
Instance:	$\{(b, w_1, \ldots, w_n) \mid b, w_1, \ldots, w_n \in \mathbb{N}\}$
Přípustná řešení:	$sol(b, w_1, \ldots, w_n) = \{C \subseteq \{1, \ldots, n\} \mid \sum_{i \in C} w_i \leq b\}$
Cena řešení:	$cost(C, (b, w_1, \ldots, w_n)) = \sum_{i \in C} w_i$
Cíl:	maximum

Idea algoritmu je následující. Pro $k \leq n$ nejdříve vygenerujeme všechny podmnožiny množiny $\{1, \ldots, n\}$ do velikosti k, takové, že pro každou z nich je suma odpovídajících prvků menší než b. V druhé fázi algoritmu se každou z podmnožin pokusíme rozšířit tak, že budeme žravým způsobem přidávat další prvky $\{1, \ldots, n\}$. Vybereme takovou rozšířenou množinu, suma jejíchž prvků je největší.

Algoritmus 5 Kombinace backtrackingu a greedy přístupu pro úlohu batohu

```
 1: procedure GenerateCandidates(⟨a_1, ..., a_i⟩, (b, w_1, ..., w_n), k)
 2:     if Σ_{i∈Set(⟨a_1,...,a_i⟩)} w_i > b then
 3:         return ∅
 4:     end if
 5:     if |Set(⟨a_1, ..., a_i⟩)| = k or i = n then
 6:         return {Set(⟨a_1, ..., a_i⟩)}
 7:     end if
 8:     X ← GenerateCandidates(⟨a_1, ..., a_i, 1⟩, (b, w_1, ..., w_n), k)
 9:     Y ← GenerateCandidates(⟨a_1, ..., a_i, 0⟩, (b, w_1, ..., w_n), k)
10:     return X ∪ Y
11: end procedure
12:
13: procedure ExtendCandidate(C, (b, w_1, ..., w_n))
14:     Vytvoř přiritní frontu Q z prvků 1, ..., n uspořádanou sestupně podle odpovídajících prvků w_1, ..., w_n
15:     while Q není prázdná do
16:         Odeber z Q prvek x
17:         if x ∉ C and Σ_{i∈C} +w_x ≤ b then
18:             C ← C ∪ {x}
19:         end if
20:     end while
21:     return C
22: end procedure
23:
24: procedure KnapsakScheme((b, w_1, ..., w_n), k)
25:     X ←GenerateCandidates(⟨⟩, (b, w_1, ..., w_n), k)
26:     B ← ∅
27:     for x ∈ X do
28:         A ←ExtendCandidate(x, (b, w_1, ..., w_n))
29:         if Σ_{x∈A} w_x > Σ_{x∈B} w_x then
30:             B ← A
31:         end if
32:     end for
33:     return B
34: end procedure
```

Složitost algoritmu závisí mimo velikosti vstupní instance i na volbě k. Pokud $k = 0$ je KnapsackScheme jenom obyčeným greedy algoritmem se složitostí $O(n \log n)$. Situace je podobná, pokud uvažujeme k jako pevně danou konstantu a počítáme složitost algoritmu závislou jenom na n. Počet kandidátů je pak totiž roste polynomicky,

jejich počet je vždy omezen $O(n^k)$. (kandidáty totiž počítáme jako k prvkové variace bez opakování) a tedy složitost celého algoritmu je polynomická v závisloti na n. Z toho také plyne, že v závislosti na k roste složitost algoritmu exponenciálně.

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.