KMI/ALGO3 — Algoritmy 3

9. Systematické prohledávání

Jan Konečný

Katedra informatiky, PřF UP Olomouc

Kde jsme: sylabus KMI/ALGO3

- **Co znamená navrhnout algoritmus** — specifikace, invariant, složitost
- **Hladové algoritmy** — kdy je lokální volba bezpečná
- **Hladové algoritmy na grafech** — kostry, nejkratší cesty
- **Rozděl a panuj** — rekurence, Master theorem
- **Rozděl a panuj II** — chytré rozklady
- **Dynamické programování I** — hledání stavů
- **Dynamické programování II** — vícerozměrné stavy
- **Toky v sítích** — max-flow/min-cut jako návrhový nástroj
- **Systematické prohledávání** — hrubá síla a backtracking
- **Branch-and-bound** — meze přesného řešení, P a NP
- **Adversariální návrh** — minimax, alfa-beta
- **Když přesné optimum není realistické** — aproximace, heuristiky
- **Randomizace a syntéza** — jak si vybrat metodu

Když neznáme zkratku

Budeme hledat způsob, jak projít exponenciální množinu kandidátů tak, abychom žádné řešení nevynechali, žádnou část prostoru neopakovali a odmítali slepé větve co nejdříve.

Hrubá síla se obvykle uvádí jako poslední možnost. Z pohledu návrhu je ale důležitější, že poskytuje úplný výchozí postup, vůči němuž měříme účinek lepší reprezentace, prořezávání a heuristik.

Postupnými úpravami z ní odvodíme backtracking a algoritmus DPLL.

Od kandidátů ke stromu stavů

Řešení vzniká posloupností rozhodnutí $x_1, \ldots, x_n$; každé x_i z konečné množiny možností.

Definice 1 (Strom stavového prostoru)

Zakořeněný strom, jehož uzly reprezentují *částečná* řešení. Potomci uzlu jsou všechna povolená rozšíření o jeden krok.

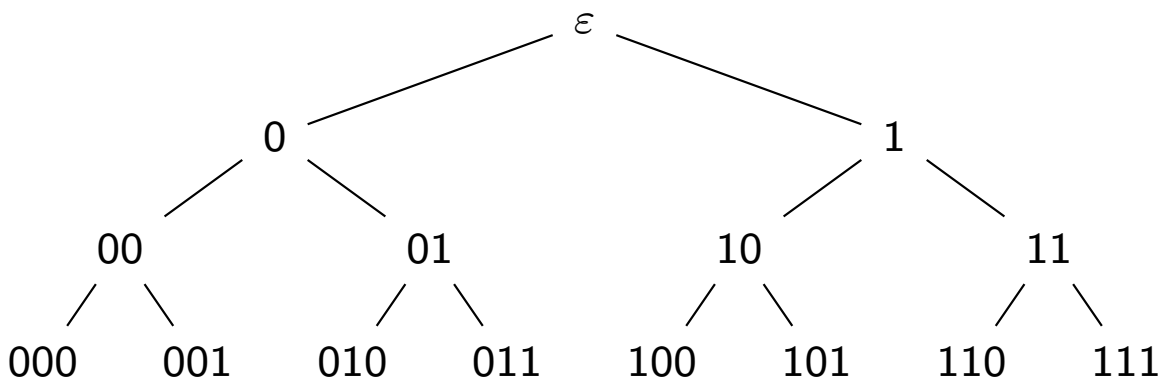
- kořen = prázdné rozhodnutí;
- uzel v hloubce k = prefix $(x_1, \ldots, x_k)$;
- hrana = jedno další rozhodnutí;
- list v hloubce n = úplný kandidát.

Reprezentace odděluje dvě otázky:

- Úplnost — leží každý kandidát v některém listu? *(Nesmíme obětovat.)*
- Efektivita — kolik uzlů musíme skutečně navštívit? *(Chceme zlepšit.)*

Tři binární rozhodnutí

Podmnožinu $\{a, b, c\}$ popíšeme bity (x_a, x_b, x_c):



Osm listů = osm podmnožin; list 101 znamená $\{a, c\}$.

Prohledávání do hloubky potřebuje pamatovat jen *aktuální cestu* — samotný strom nikdy explicitně nevytváříme.

Hrubá síla jako korektní základ

Generuj a otestuj

1. Systematicky vygeneruj všechny úplné kandidáty.
2. Každý otestuj proti všem podmínkám.
3. První nalezené řešení vrať (nebo zaznamenej všechna).
4. Nevyhoví-li žádný, ohlas neexistenci.

Má-li každý z n kroků b možností, vzniká b^n listů; s testem za $T(n)$ je mez $O(b^n T(n))$.

Věta 2

Vytvoří-li generátor každý kandidát alespoň jednou a přijme-li test právě platná řešení, nalezne „generuj a otestuj“ řešení právě tehdy, když existuje.

Korektnost má *dvě oddělené* části: test musí být spolehlivý, generátor úplný.

Dva základní generátory

Podmnožiny(i, S)

Je-li $i = n + 1$, zpracuj S. Jinak zavolej $\textsc{Podmnožiny}(i + 1, S)$ a $\textsc{Podmnožiny}(i + 1, S \cup \{a_i\})$.

2^n listů, žádné duplicity (podmnožina $\leftrightarrow$ binární vektor), paměť $O(n)$.

Permutace(P, U)

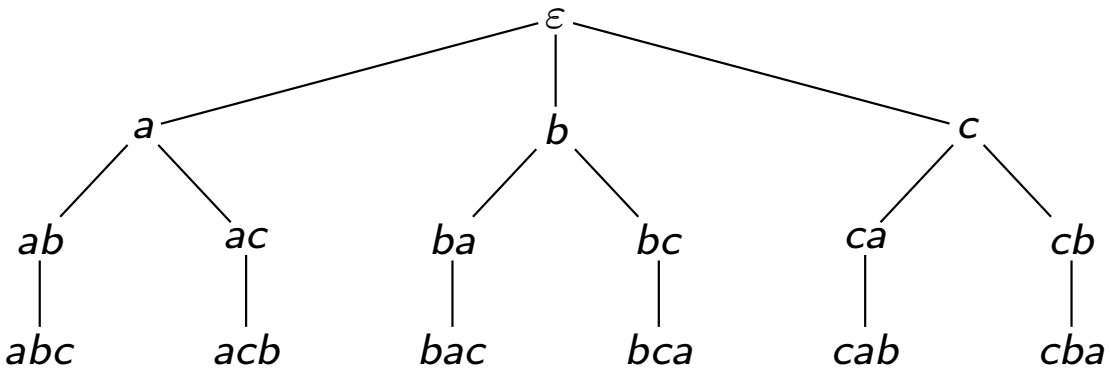
Je-li $U = \emptyset$, zpracuj P. Jinak pro každý $x \in U$:

1. připoj x na konec P;
2. zavolej $\textsc{Permutace}(P, U \setminus \{x\})$;
3. odstraň x z konce P.

$n!$ listů. Třetí krok je typický princip backtrackingu: proveď změnu — prozkoumej větev — změnu vrať.

Permutace: strom a jeho velikost

Pro $\{a, b, c\}$:



- Větvení se *zmenšuje*: n, pak $n-1$, ... Listů je $n!$.
- Uzlů je $\sum_{k=0}^{n} \frac{n!}{(n-k)!} \approx e \cdot n!$ — jen o konstantu víc než listů.
- U podmnožin byl poměr uzlů k listům ≈ 2, u permutací $\approx e$. Vždy se vyplatí spočítat, kolik uzlů strom má.

$8! = 40\,320$, $12! \approx 4{,}8 \cdot 10^8$, $20! \approx 2{,}4 \cdot 10^{18}$ — hrubá síla nad permutacemi končí kolem $n = 12$.

Backtracking: testuj už prefix

Úplný výčet kontroluje omezení až v listu. Často však už *prefix* dokazuje, že žádné jeho pokračování nemůže být řešením.

Definice 3 (Bezpečný ořez)

Predikát $\textsc{Slibný}(p)$ umožňuje bezpečný ořez, jestliže

$$\neg\textsc{Slibný}(p) \implies \text{žádné rozšíření prefixu } p \text{ není řešením.}$$

Obecný backtracking(p)

1. Porušuje-li p podmínku nutnou pro každé řešení, vrať se.
2. Je-li p úplné řešení, zpracuj je a vrať se.
3. Zvol další nerozhodnutou proměnnou.
4. Pro každou přípustnou hodnotu v: proveď rozhodnutí, rekurzivně pokračuj, rozhodnutí vrať.

Korektnost a nejčastější chyba

Věta 4

Pokrývají-li děti uzlu všechna jeho možná rozšíření a je-li každý ořez bezpečný, nalezne backtracking všechna řešení dosažitelná z kořene.

Důkaz.

Rozhodnutí libovolného řešení s určují cestu od kořene k listu. Žádný prefix té cesty nesmí být ořezán — kdyby byl, měl by rozšíření s, což odporuje bezpečnosti ořezu. Naopak zpracujeme jen listy, které prošly závěrečnou kontrolou.

Nejčastější chyba

„Tato větev vypadá špatně“ není důkaz. Prořezávací pravidlo musí vyjadřovat *nutnou podmínku*; heuristika smí měnit pořadí větví, nikoli svévolně větev zahazovat.

Případová studie: n dam

Naivně bychom vybírali n polí z n^2. Struktura omezení ale nabízí lepší reprezentaci: v každém řádku právě jedna dáma. Stav $(q_1, \ldots, q_k)$ udává sloupce dam v prvních k řádcích.

Dámu v řádku $k+1$, sloupci c lze přidat právě tehdy, když pro všechna $1 \leq i \leq k$

$$c \neq q_i \qquad \text{a} \qquad |c - q_i| \neq k + 1 - i.$$

První podmínka zakazuje společný sloupec, druhá společnou diagonálu.

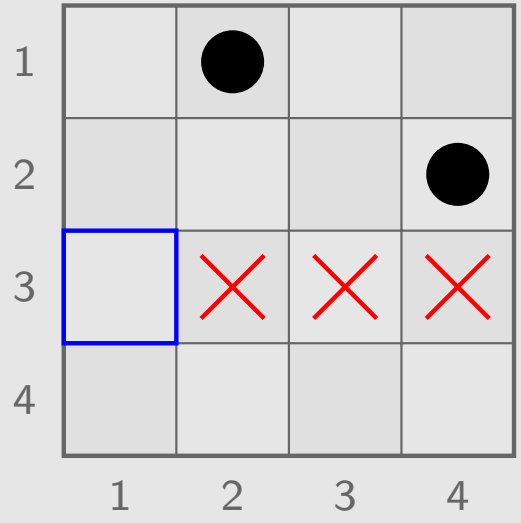
Lemma 5 (Bezpečnost ořezu)

Ohrožují-li se dvě dámy v prefixu, žádné jeho rozšíření není řešením.

Důkaz. Rozšíření pouze přidává další dámy; polohy již umístěných nemění, takže konfliktní dvojice zůstane konfliktní.

Čtyři dámy: prefix $(2, 4)$

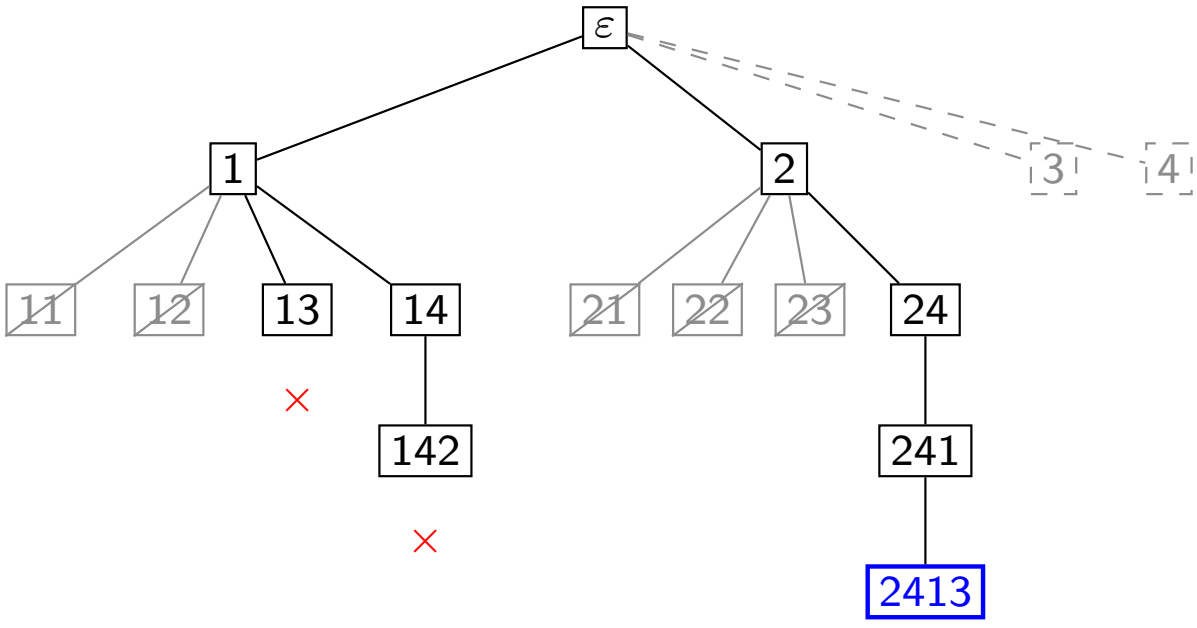
Příklad 6



Ve třetím řádku je jediným bezpečným sloupcem 1; prefix $(2, 4, 1)$ rozšíříme sloupcem 3 a dostaneme řešení $(2, 4, 1, 3)$.

Stačí udržovat tři množiny: obsazené sloupce c, diagonály $r - c$ a $r + c$. Jako booleovská pole je test i vrácení tahu $O(1)$.

Strom backtrackingu pro $n = 4$



Šedě přeškrtnuté děti ořízne test sloupců a diagonál okamžitě, × značí uzel bez bezpečného rozšíření, modrý list je řešení. Podstromy pro $q_1 = 3, 4$ jsou zrcadlové. Z $4^4 = 256$ listů prozkoumáme jen hrstku uzlů.

Počet řešení úlohy n dam

n	4	5	6	7	8	9	10	11	12
řešení	2	10	4	40	92	352	724	2680	14200
$n!$	24	120	720	5040	40320			...	

- Pro $n = 2, 3$ řešení *neexistuje* (snadno se ověří ručně), pro každé $n \geq 4$ existuje.
- Posloupnost je nepravidelná ($n = 6$ má jen 4 řešení, méně než $n = 5$!) a uzavřený vzorec pro počet řešení není znám.
- Známé hodnoty končí u $n = 27$ — spočteno v roce 2016 na poli FPGA obvodů za několik měsíců.

Úlohu položil roku 1848 šachista Max Bezzel; všech 92 řešení pro $n = 8$ publikoval Franz Nauck o dva roky později. Zabýval se jí i Gauss, který se zpočátku v počtu spletl.

Implementace bitovými maskami

Tři množiny (sloupce, diagonály $r - c$, diagonály $r + c$) se vejdou do tří celých čísel jako bitové masky:

`cols`	bit c = sloupec c obsazen
`d1`	posunuto o jedna *doleva* při každém sestupu
`d2`	posunuto o jedna *doprava* při každém sestupu

Volná pole aktuálního řádku jsou pak jediným výrazem

$$\texttt{free} = \neg(\texttt{cols} \mid \texttt{d1} \mid \texttt{d2}) \;\&\; \texttt{mask},$$

a nejnižší volný bit získáme výrazem $\texttt{free} \;\&\; (-\texttt{free})$.

Rekurze pak nepotřebuje žádný zápis do paměti mimo tři čísla; vrácení tahu obstará samotný návrat z volání.

Táž myšlenka (stav jako bitová maska) se objeví u Held–Karpova DP pro TSP i u moderních řešičů SAT.

Omezující podmínky a propagace

Backtracking = řešení *problému splňování omezení*: proměnné $X_1, \ldots, X_n$, domény $D_1, \ldots, D_n$, podmínky.

Definice 7 (Propagace omezení)

Odstraňuje z domén hodnoty, které už nemohou být součástí žádného řešení slučitelného s aktuálním částečným přiřazením. Prázdná doména je důkazem, že větev nemá řešení.

V Sudoku přiřazení čísla okamžitě odstraní toto číslo z domén políček ve stejném řádku, sloupci a bloku. Zbude-li políčku jediná hodnota, můžeme ji *vynutit bez větvení*.

Když propagace nestačí, zvolíme proměnnou a rozvětvíme se podle jejích zbývajících hodnot.

Krok propagace v Sudoku

Příklad 8

Políčko z: jinde v řádku už leží $3, 4, 9$, ve sloupci 8, a jeho blok:

5		1
	z	7
2		

$$D_z = \{1, \ldots, 9\} \setminus \big(\{3, 4, 9\} \cup \{8\} \cup \{1, 2, 5, 7\}\big) = \{6\}$$

Propagace vynutí $z = 6$ *bez větvení*; škrtnutí šestky v sousedství může vyvolat další dedukce.

Dvě pravidla propagace v Sudoku

V Sudoku se propagace obvykle popisuje dvěma pravidly:

- **Naked single** — políčku zbyla *jediná* hodnota v doméně. (To je přesně náš případ $D_z = \{6\}$.)
- **Hidden single** — hodnota v se v celém řádku (sloupci, bloku) vejde do *jediného* políčka, i kdyby to políčko mělo v doméně víc hodnot.

Proč obě pravidla

Hidden single z *naked single* neplyne: dívá se na jednotku zvenčí — ptá se „kam se vejde tato hodnota“, ne „co může být v tomto políčku“. Každé z pravidel odvodí něco, co druhé neodvodí.

Silnější propagace („pointing pairs“, „X-wing“…) ořeže víc větví, ale stojí víc času na uzel. K tomuto kompromisu se vrátíme na konci přednášky.

Ořez, který se odvolává na hodnoty: součet podmnožiny

Máme kladná čísla $a_1, \ldots, a_n$ a cíl B; hledáme podmnožinu se součtem přesně B.

Dva bezpečné ořezy

- **Přeplnění:** je-li součet prefixu už $> B$, žádné rozšíření nepomůže — *u kladných čísel*.
- **Nedosažitelnost:** je-li součet prefixu plus součet *všech* zbývajících čísel menší než B, také končíme.

Druhý ořez vyžaduje předpočítané *sufixové součty*; příprava je levná a ořez může odstranit velké podstromy.

Ořez, který se odvolává na hodnoty: součet podmnožiny

Máme kladná čísla $a_1, \ldots, a_n$ a cíl B; hledáme podmnožinu se součtem přesně B.

Dva bezpečné ořezy

- **Přeplnění:** je-li součet prefixu už $> B$, žádné rozšíření nepomůže — *u kladných čísel*.
- **Nedosažitelnost:** je-li součet prefixu plus součet *všech* zbývajících čísel menší než B, také končíme.

Druhý ořez vyžaduje předpočítané *sufixové součty*; příprava je levná a ořez může odstranit velké podstromy.

Se zápornými čísly oba ořezy přestanou platit: přebytek lze později vykompenzovat. Musíme pak sledovat součet záporných zbývajících čísel zvlášť — nebo se ořezu vzdát.

Bezpečnost ořezu je tedy vždy tvrzením o tom, co ještě může přijít.

Heuristika nejmenší domény a tři role pravidel

MRV (*minimum remaining values*) vybírá nerozhodnutou proměnnou s nejmenší doménou — „nejprve nejtěsnější místo“.

Korektnost i úplnost zůstávají zachovány, protože *pořadí* přiřazování proměnných smí být libovolné; množina řešení se nemění. Nejhorší exponenciální mez ovšem žádná heuristika neodstraní.

Tři role pravidel

- **Omezení** definuje, co *je* řešení.
- **Propagace a bezpečný ořez** odvozují *nemožnost* celé větve.
- **Heuristika** rozhoduje, kterou *stále možnou* větev zkusit dříve.

Volba proměnné vs. volba hodnoty

kterou proměnnou	**kterou hodnotu první**
MRV — nejmenší doména	*LCV* — hodnota, která nejméně omezí sousedy
cíl: rychle najít spor	cíl: rychle najít řešení
mění tvar stromu	mění pořadí větví

obojí zachovává úplnost — žádná větev se nezahazuje

Proč se heuristiky doplňují

Nejtěsnější proměnná má nejmenší větvení, takže případný spor se objeví blízko u kořene a ořízne velký podstrom. Naopak nejméně omezující hodnota nechá co nejvíc možností otevřených, takže první sestup má největší šanci dojít až k řešení.

Ani jedno není záruka: existují instance, na nichž obě heuristiky vedou hůř než náhodné pořadí. V praxi však obvykle výrazně pomáhají.

Symetrie

Mnoho úloh má řešení, která jsou „táž až na přeznačení“. Backtracking je ale prochází *všechna*.

- *n* **dam.** Každé řešení má až 8 obrazů (4 rotace × zrcadlení). Omezíme-li dámu v prvním řádku na levou polovinu, ušetříme skoro polovinu práce.
- **Barvení grafu.** Barvy jsou zaměnitelné — první vrchol smíme napevno obarvit barvou 1, jeho souseda barvou 2 atd.
- **Rozdělení do skupin.** Skupiny bez jmen: vynutíme, aby nejmenší prvek každé skupiny rostl.

Pozor na formulaci

Lámání symetrie je bezpečné jen tehdy, když z každé třídy ekvivalentních řešení ponecháme alespoň jedno. Hledáme-li *všechna* řešení, musíme je nakonec zase rozgenerovat.

SAT jako prohledávání ohodnocení

Formule je splnitelná, existuje-li ohodnocení, pro které je pravdivá.
Pracujeme v CNF: konjunkce klauzulí, klauzule je disjunkce literálů.

Příklad 9

$$F = (x \vee \neg y) \wedge (y \vee z) \wedge (\neg x \vee \neg z)$$

Ohodnocení $x = 1, y = 1, z = 0$ splňuje všechny tři klauzule.

Proč prohledávání? SAT je NP-úplný — historicky vůbec první problém, u něhož byla NP-úplnost dokázána (Cookova věta; blíže příště).
Polynomiální algoritmus nikdo nezná.

Proto je důležité prohledávat efektivně.

Částečné ohodnocení a ořez

Hrubá síla zkusí všech 2^n úplných ohodnocení. Backtracking pracuje s *částečným* ohodnocením; klauzule může být:

- splněná některým pravdivým literálem;
- nerozhodnutá — obsahuje nepřiřazenou proměnnou;
- sporná — všechny její literály jsou nepravdivé.

Sporná klauzule je bezpečný důvod k ořezu: další hodnoty už přiřazených proměnných ji nezmění.

DPLL: větvení doplněné dedukcí

Definice 10 (Jednotková propagace)

Má-li nerozhodnutá klauzule jediný dosud neurčený literál, musí být tento literál pravdivý — hodnotu tedy *vynutíme*.

V $(x \vee \neg y \vee z)$ po $x = 0, y = 1$ musí platit $z = 1$. Každé řešení slučitelné s prefixem má tuto hodnotu, nejde tedy o heuristiku.

Definice 11 (Čistý literál)

Literál je čistý, nevyskytuje-li se jeho negace v žádné dosud nesplněné klauzuli. Čistý literál můžeme bez větvení ohodnotit jako pravdivý.

Rozdíl: hodnota čistého literálu *není vynucená* (jiné řešení ji může mít opačnou), ale *splnitelnost se zachovává*, a pro hledání jednoho řešení to stačí. (Davis, Logemann, Loveland 1962.)

Algoritmus DPLL

DPLL(F, A)

1. Opakuj ve F jednotkovou propagaci (příp. i čisté literály).
2. Je-li některá klauzule sporná, vrať NESPLNITELNÁ.
3. Jsou-li všechny klauzule splněné, vrať A doplněné libovolně.
4. Zvol nepřiřazenou proměnnou x.
5. Zkus DPLL s jednou hodnotou x; uspěje-li, vrať výsledek.
6. Jinak zkus DPLL s opačnou hodnotou x a vrať jeho výsledek.

Příklad 12 (Řetěz jednotkové propagace)

$$(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c)$$

Poslední klauzule vynutí $c = 0$, druhá $b = 0$, první $a = 0$ — bez jediného větvení.

DPLL krok za krokem

$$F = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (a \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee \neg a)$$

Žádná jednotková klauzule — musíme větvit, a to podle a; nejprve zkusíme $a = 1$.

DPLL krok za krokem

$$F = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (a \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee \neg a)$$

Žádná jednotková klauzule — musíme větvit, a to podle a; nejprve zkusíme $a = 1$.

Větev $a = 1$:

klauzule	stav	vynucuje
$(\neg a \vee b)$	$\neg a$ nepravdivé	$b = 1$
$(\neg b \vee c)$	$\neg b$ nepravdivé	$c = 1$
$(\neg c \vee d)$	$\neg c$ nepravdivé	$d = 1$
$(\neg d \vee \neg a)$	oba literály nepravdivé	SPOR

Větev skončí sporem jen díky propagaci, *bez dalšího větvení*.

DPLL krok za krokem

$$F = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (a \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee \neg a)$$

Žádná jednotková klauzule — musíme větvit, a to podle a; nejprve zkusíme $a = 1$.

Větev $a = 1$**:**

klauzule	stav	vynucuje
$(\neg a \vee b)$	$\neg a$ nepravdivé	$b = 1$
$(\neg b \vee c)$	$\neg b$ nepravdivé	$c = 1$
$(\neg c \vee d)$	$\neg c$ nepravdivé	$d = 1$
$(\neg d \vee \neg a)$	oba literály nepravdivé	SPOR

Větev skončí sporem jen díky propagaci, *bez dalšího větvení*.

Větev $a = 0$: $(a \vee c)$ vynutí $c = 1$, $(\neg c \vee d)$ vynutí $d = 1$. Klauzule $(\neg a \vee b)$ i $(\neg d \vee \neg a)$ jsou splněné literálem $\neg a$, $(\neg b \vee c)$ literálem c.

Zbývá volné b; formule je tedy splnitelná, například $a = 0, b = 0, c = 1, d = 1$.

Nesplnitelná formule

$$F = (x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$$

Čtyři klauzule zakazují všechna čtyři ohodnocení dvou proměnných.

větev	propagace	výsledek
$x = 0$	$(x \vee y)$ vynutí $y = 1$; $(x \vee \neg y)$ vynutí $y = 0$	spor
$x = 1$	$(\neg x \vee y)$ vynutí $y = 1$; $(\neg x \vee \neg y)$ vynutí $y = 0$	spor

Obě větve selhaly, a protože pokrývají všechny možnosti, je F nesplnitelná.

Asymetrie SAT a UNSAT

SAT doložíme krátkým certifikátem — stačí ukázat ohodnocení.

UNSAT vyžaduje důkaz, že *žádné* ohodnocení nefunguje. Zde jsme prošli obě větve; obecně je takový důkaz mnohem delší. Tato asymetrie se odráží v definici třídy NP, která požaduje krátké certifikáty jen pro *kladné* odpovědi.

Korektnost DPLL

Věta 13

DPLL vrátí splňující ohodnocení právě tehdy, když je vstupní formule splnitelná.

Důkaz.

Jednotková propagace zachovává splnitelnost: jediný nerozhodnutý literál jednotkové klauzule musí být pravdivý v každém splňujícím rozšíření. Spornou klauzuli už žádné rozšíření neopraví. Nelze-li nic odvodit, pokrývají obě větve ($x = 0$, $x = 1$) všechny možnosti. Indukcí podle počtu nepřiřazených proměnných dostáváme úplnost i spolehlivost.

V nejhorším případě musí DPLL projít strom s 2^n listy. V praxi však propagace a vhodná volba větvící proměnné strom pro mnohé instance výrazně zmenší.

Moderní řešiče SAT přidávají zejména učení klauzulí a řízené návraty; základní schéma větvení a dedukce je stejné jako v DPLL.

Od DPLL ke CDCL

Když DPLL narazí na spor, vrátí se o *jednu* úroveň. Často je to málo: spor mohla způsobit proměnná přiřazená mnohem dřív.

CDCL — *conflict-driven clause learning*

1. Ze sporu se odvodí nová klauzule, která jej vysvětluje (analýza implikačního grafu).
2. Klauzule se *přidá* k formuli — od té chvíle propagace tentýž spor předejde.
3. Algoritmus skočí zpět rovnou na úroveň, kde má nová klauzule smysl (*non-chronological backjumping*).

- Naučené klauzule jsou logickými *důsledky* vstupu, takže splnitelnost se nemění — korektnost zůstává.
- K tomu *restarty*, heuristika VSIDS a datová struktura *two watched literals*.

Praktický rozmach řešičů SAT začal programy Chaff (2001) a MiniSat (2003).

NP-úplný problém, který se běžně řeší

SAT je NP-úplný, přesto dnešní řešiče běžně zvládají formule s miliony proměnných.

- **Verifikace hardwaru.** Ekvivalence dvou obvodů se převede na SAT; Intel i ostatní výrobci to používají od aféry s chybou dělení v Pentiu (1994).
- **Ohraničené ověřování modelu.** „Může software během k kroků dosáhnout chybového stavu?“ $\rightarrow$ jedna velká formule.
- **Plánování a rozvrhy.** Zakódování do SAT bývá rychlejší než psaní vlastního řešiče.
- **Matematické důkazy.** Booleovský pythagorejský problém (2016) — důkaz o velikosti 200 TB, vygenerovaný řešičem SAT.
- **Balíčkovací systémy.** Řešení závislostí balíčků je úloha typu SAT; např. `conda` k ní používá řešič SAT.

NP-těžkost mluví o nejhorším případě. Reálné instance mají strukturu a propagace ji umí využít.

Cena prohledávání

Čas neurčuje počet *řešení*, nýbrž počet navštívených *částečných stavů*. S N navštívenými uzly, cenou C na uzel a hloubkou d:

$$\text{čas } O(NC), \qquad \text{paměť DFS } O(d + S),$$

kde S je velikost měnitelného stavu. Hodnota N závisí na instanci, pořadí větví i síle propagace.

Co porovnávat experimentálně

Nestačí měřit jen čas. Sledujeme také *počet navštívených uzlů*, *počet ořezů*, *průměrnou hloubku sporu* a *náklady propagace*.

Silnější test může navštívit méně uzlů, ale být v každém uzlu dražší.

Doplněk: číslování permutací

Někdy nechceme permutace *generovat*, ale umět mezi nimi „skočit“ — třeba rozdělit práci mezi procesory. Pomůže faktoradická soustava.

Číslo $k < n!$ zapíšeme jako

$$k = c_{n-1} \cdot (n-1)! + c_{n-2} \cdot (n-2)! + \cdots + c_1 \cdot 1!, \qquad 0 \leq c_i \leq i.$$

Číslice c_i pak čteme jako „ber c_i-tý dosud nepoužitý prvek“.

Příklad 14

$k = 13$, $n = 4$: $13 = 2 \cdot 3! + 0 \cdot 2! + 1 \cdot 1!$, tedy číslice $(2, 0, 1)$.

Ze seznamu (a, b, c, d): ber 2. $\to c$; zbývá (a, b, d), ber 0. $\to a$; zbývá (b, d), ber 1. $\to d$; zbývá b. Výsledek (c, a, d, b).

Prohledávání i číslování používají tentýž strom; jednou jím sestupujeme rekurzivně, podruhé podle číslic.

Počty uzlů pro n dam

Srovnání počtu *navštívených uzlů* podle síly testu:

n	bez ořezu	jen sloupce	sloupce + diagonály
6	55 987	1 957	153
8	19 173 961	109 601	2 057
10	$\approx 1{,}1 \cdot 10^{10}$	9 864 101	35 539

(Všechny uzly včetně kořene při hledání všech řešení.)

- Zákaz opakovat sloupec ubral *exponent*: z n^n na $n!$.
- Diagonální test ubral zhruba další dva řády — ale nejhorší případ zůstává exponenciální.

Prořezávání tedy mění konstanty a základ exponentu, ne třídu složitosti.

Jak navrhnout backtracking

1. **Zvol rozhodnutí.** Z jakých kroků se skládá úplné řešení?
2. **Definuj stav.** Jaké minimum informace popisuje prefix?
3. **Dokaž úplnost větvení.** Pokrývají děti všechny možnosti?
4. **Najdi nutné podmínky.** Co už žádné pokračování neopraví?
5. **Udržuj změny lokálně.** Co přidat před rekurzí a vrátit po ní?
6. **Až potom přidej heuristiku.**

Backtracking není branch-and-bound

Backtracking zahazuje větev, protože nemůže obsahovat *přípustné* řešení.

Branch-and-bound (příště) zahazuje i *přípustnou* větev, protože pomocí meze dokáže, že nemůže zlepšit dosud nejlepší *hodnotu*.

Shrnutí

- Úplný výčet je korektní základ, ne automaticky dobrá implementace.
- Strom stavového prostoru organizuje kandidáty pomocí prefixů.
- Bezpečný ořez musí být podložen *nutnou podmínkou*.
- „Proveď–zanoř–vrať“ umožňuje sdílet jeden pracovní stav.
- Propagace odvozuje *vynucená* rozhodnutí; heuristika jen řadí možnosti.
- DPLL spojuje větvení, detekci sporu a jednotkovou propagaci.
- Lepší prořezávání často výrazně pomůže, ale nejhorší případ může zůstat exponenciální.

Literatura


M. Davis, G. Logemann, D. Loveland.
A Machine Program for Theorem-Proving.
CACM 5 (1962), 394–397.


S. Dasgupta, C. Papadimitriou, U. Vazirani.
Algorithms. McGraw–Hill (2006).


J. Kleinberg, É. Tardos.
Algorithm Design. Addison–Wesley (2006).


T. H. Cormen a kol.
Introduction to Algorithms. MIT Press (2001).