3 Exaktní metody a heuristiky

Podle prohledávání stavového prostoru dělíme metody na:
úplné (všechny stavy), systematické (každý stav max. 1), obojí = exaktní řešení

3.1 Lokální metody, Pojem okolí v lokálních metodách, Konstruktivní a iterativní heuristické metody, Obrana před uváznutím v lokálním minimu

Stav Nechť $X = \{x_1, x_2, \ldots, x_n\}$ jsou konfigurační proměnné problému Π. Nechť $Z = \{z_1, z_2, \ldots, z_m\}$ jsou vnitřní proměnné algoritmu A řešícího instanci I problému Π. Pak každé ohodnocení s proměnných $X \cup Z$ je *stav* algoritmu A řešícího I.

Stavový prostor Nechť $S = \{s_i\}$ je množina všech stavů algoritmu A řešícího I. Nechť $Q = \{q_j\}$ je množina operací $S \to S$ takových, že $q_j(s_i) \neq s_i$ pro všechna s_i, q_j. Pak dvojici (S, Q) nazveme *stavovým prostorem* algoritmu A řešícího I.

Okolí stavu Okolí stavu $s \in S$ je množina stavů, dosažitelných z s aplikací některé operace $q \in Q$.

k-okolí stavu k-okolí stavu $s \in S$ je množina stavů, dosažitelných z s aplikací nejméně jedné a nejvýše k operací $q \in Q$.

Sousední stav Stav $s_0 \in S$ patřící do okolí stavu $s \in S$ se nazývá *sousedním stavem* (sousedem) stavu s.

Lokální metoda má vždy jen jeden aktuální stav, který zpracovává, přičemž uvažuje sousední stavy z (relativně malého) okolí. *Úplné* a *systematické* metody uschovávají každý stav z okolí pro pozdější zpracování. Proto jsou exaktní a proto mohou mít nadpolynomiální složitost.

Konstruktivní heuristika Začne z triviální konfigurace a postupnými kroky konstruujeme řešení.

Iterativní heuristika Začne z nějakého řešení a to postupně vylepšuje.

Dvojfázové heuristiky První fáze slouží k získání řešení (konstruktivně, náhodné řešení), ve truhé fázi iterativní vylepšování.

Jednoduché heuristiky

- Pouze nejlepší
- Prvé zlepšení
- Heuristiky Kernighan-Lin (mají takové to podlouhlé okolí, projdou například pět stavů a pak se podívají, který z těch pěti byl nejlepší a z něj pokračují)

Lokální heuristiky, které neřeší problém lokálních minim, uváznou hned v prvním takovém. To se projevuje tím, že výsledné řešení silně závisí na startovacím stavu. Existuje několik metod, jak uváznutí řešit. Můžeme

- zvětšit okolí,
- startovat z několika různých (náhodných) počátečních řešení,
- vracet se z větví, které nevedou k řešení (detekujeme slepou uličku),
- dočasně připustit zhoršení aktuálního stavu (zvyšuje nároky na řízení algoritmu)

3.2 Globální metody

Při řešení instancí problému globálními metodami si pomáháme dekompozicí instance na podinstance. Výsledné řešení skládáme z jednotlivých řešení podinstancí. Triviální podinstance řešíme hrubou silou. Přirozeným modelem výpočtu je rekurze.

Základní postup tedy vypadá nějak takto:

$$I \in \Pi \left\{ \begin{array}{lcl} I_1 \in \Pi & \longrightarrow & Y_1 \\ I_2 \in \Pi & \longrightarrow & Y_2 \end{array} \right\} Y$$

instanci problému Π rozložíme na menší instance problému Π a vyřešíme je. Získaná řešení Y_1 a Y_2 pak sloučíme do výsledného řešení Y původní instance problému Π.

Podle specifických vlastností lze dekompozice rozdělit do tří skupin.

Přibližná dekompozice Pokud jsou Y_1 resp. Y_2 optimálními řešeními instancí I_1 resp. I_2, pak je Y řešením (ne nutně optimálním) instance I. Pokud Y_1, Y_2 neexistují, nevíme nic.

Čistá dekompozice Pokud jsou Y_1 resp. Y_2 optimálními řešeními instancí I_1 resp. I_2, pak je Y optimálním řešením instance I. Pokud Y_1, Y_2 neexistují, Y také neexistuje.

Přesná dekompozice Taková čistá dekompozice, že každé optimální řešení Y se dá složit z nějakých optimálních řešení Y_1 a Y_2 (ze všech optimálních řešení Y_1, Y_2 se dají složit všechna optimální řešení Y.

Pokud je navíc dekompozice taková, že jedna z dekomponovaných instancí je triviální, hovoříme o *redukci* (přibližné, čisté, přesné).

Dekompozice:

- optimlální I1..Ik => suboptimální I
- čistá: optimlální I1..Ik => optimální I (pokud I nemá řešení, alespoň jeden In taky ne)
- přesná: optimlální I1..Ik => všechna optimální I

Dílčí řešení jsou optimální, páč jinak se chyba kumuluje.

3.3 Rekurze

Rekurze znamená *sebeopakování*

Def: Nechť f je funkce n proměnných a funkce g n+2 proměnných. Operací primitivní rekurze nazveme funkci r(x,y), jestliže pro všechna x a pro y=[y1,...,yn] platí

1. r(0,y) = f(y)
2. r(x+1,y) = g(x, y, r(x,y))

PRF - třída primitivně rekurzních fcí je uzávěr množiny R={S(x), N(x), $U_i^n(x)$} nad operacemi skládání a primitivní rekurze. Třída PRF zahrnuje všechny numerické fce, se kterými se setkáváme. Např.: x+y, x.y, y^x, x!.

Rekurzivní podprogram - podprogram Q je rekurzivní, když používá podprogramy, z nichž alespoň jeden je původním podprogramem Q. Takové použití podprogramu naz. rekurzivním voláním podprogramu. Speciálním případem je přímé rekurzivní volání, tj. případ. kdy v operační části programu dochází k volání téhož podprogramu.

Substituční metoda - odhadneme řešení a matematickou indukcí je dokážeme.

Iterační metoda - spočívá v převedení rekurentního výrazu T(n) v řadu, kterou pak sečteme.

Mistrovská metoda - podle tvaru rekurentní formule vybereme známé řešení.

$T(n)=a.T(n/b)+f(n)$

$T(n)=\Theta(n^k)$, jestliže $f(n)=O(n^{k-E})$, $E>0$

$T(n)=\Theta(n^k.\log n)$, jestliže $f(n)=\Theta(n^k)$

$T(n)=\Theta(f(n))$, jestliže $f(n)=\Omega(n^{k+E})$, $E>0$ a když $a.f(n/b) \leq c.f(n)$ pro $c<1$ a n velké

Strom rekurze - kořenový strom, v němž každý uzel reprezentuje jeden výpočet algoritmu, vzdálenost uzlu od kořene udává hloubku rekurzivního volání.

Převod rekurze na iteraci - v iteračním programu jsou třeba navíc tzv. akumulační proměnné, v nichž se hodnoty z předchozí iterace přepisují novými hodnotami. Ne vždy ovšem vystačíme s jednou proměnnou ve fci akumulátoru. Počet pomocných proměnných můžeme stanovit pomocí tzv. grafu závislosti.

implementace

při kazdem vnoreni se na zasobniku vyhradi pamet pro datovy segment (lok. promenne) a navratova adr. Na konci procedury se provede navrat na adresu ret, ukazatel se nastavi na zacatek stavovych promennych pod vrcholem zasobniku. Vsechny lokalni promenne se adresuji relativne vzhledem k ukazateli na datovy segment procedury, která se prave provadi.

dokazte, ze soucet je rekurzivni funkce

A(x,y)=x+y je PRF neboť A(0,y)=0+y=y; A(x+1,y)=(x+1)+y=(x+y)+1=A(x,y)+1

jak zjistit slozitost rekurze

Pametova slozitost $Sp=h*L_S+L_G$. Při vypoctu oper sloz ci cas sloz T(n) vychazime z toho, ze v kazde nizsi hladine rekurze se rozsah reseneho problemu snizuje. Napr. Merge sort T(n)=O(1) n=1; 2*T(n/2)+O(n). Dále substitucni, iteracni a mistrovskou metodou.

Mistrovska: T(n)=aT(n/b)+f(n), kde a,b jsou konst a f(n) je asympt. pozit. funkce.

mechanizmus detekce rekurze při prekladu

je třeba zkoumat strukturu volani celeho programu (SVCP), napr. Sestrojenim relace $\rho \subseteq PxP$, $\rho=\{[Pi,Pj]$: Pi vola proceduru Pj$\}$, kde P je mnozina všech procedur. Proc. Pi je rekurzivni, když $[Pi,Pi] \in \rho^*$, neboli když strukt SVCP vyjadrime orientovanym grafem G, kde uzly budou reprezentovat procedury a hrany volani proc, pak Pi bude rekurze volana, jestlize lezi v nekterem z cyklu grafu G.

3.4 Iterace

Iterace v programování znamená opakované volání funkce v počítačovém programu. Zvláštní formou iterace je rekurze.

Ukázka iterace:

```
var i, a := 0          // inicializace před iterací
for i from 1 to 3 {    // smyčka se opakuje třikrát
    a := a + i         // zvýšit hodnotu o hodnotu i
}
print a                // vytisknout číslo 6
```

V tomto úseku programu se hodnota proměnné i postupně mění a nabývá hodnot 1, 2 a 3. Taková postupná změna je charakteristickým znakem iterace.

3.5 Rozděl a panuj

Algoritmy rozděl a panuj jsou založeny na přibližné dekompozici. Opakované řešení dekomponovaných instancí je řídké. Zvláštním případem jsou metody zmenši a panuj, kdy je potřeba řešit jen jednu z dekomponovaných instancí.

Příklady: Quick sort, Merge sort

Předpoklady: Rovnoměrné rozdělení na k podprostorů, sloužením jednotlivých řešení V(i) dostaneme výsledné řešení V(n).

1. Rozděl prostor P(n) na prostory P(i), Sjednocení P(i) je P(n)
2. Aplikuj algoritmus A na podprostory P(i) a dostaneš řešení V(i)
3. Sloučením V(i) vznikne řešení V(n)

Oper. slož. $T(n)=T_{Rozdělení} + kT(n/k) + T_{Sloučení}$

3.6 Algoritmy pro prohledávání stavového prostoru

3.6.1 S návratem

Pro grafové algoritmy – stromy

prohledávám do hloubky, když uzel nevyhovuje nějaké podmínce, vrátím se a jedu v jiném následníkovi = prohledávání do hloubky (programuje se rekurzí).

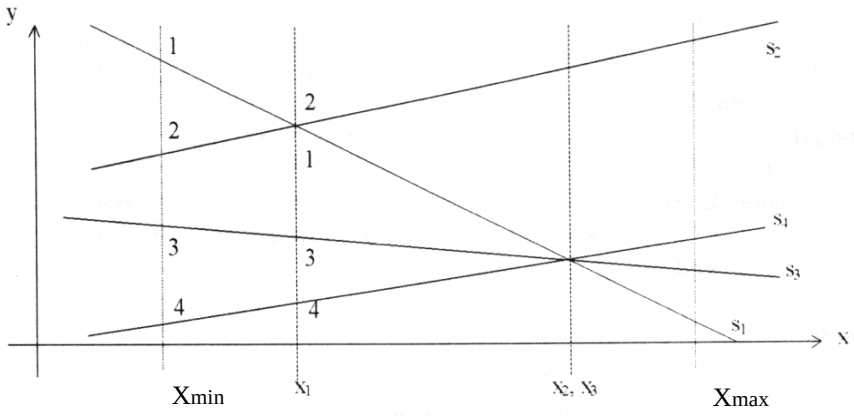
Označuje se jako algoritmus „POKUS OMYL“ a řeší se jím problém pokrytí, vzájemného vyloučení (šachovnice 8 dam), optimálního výběru a problém vzájemného přiřazení.

3.6.2 Metoda větví a hranic)

Odřezávají se neperspektivní větve na základě znalostí optimalizačního kritéria a omezujících podmínek.
(Jedná se o jistou verzi prohledávání do šířky.)
Příklad:
Máme funkci, kterou chceme maximalizovat a omezující podmínky pro hledané řešení.
Hledáme konfiguraci (ohodnocení vstupních veličin), která je maximální a vyhovuje omezujícím podmínkám.
Princip:
Budeme vytvářet strom stavového prostoru od kořene. Jednotlivé uzly jsou konfigurací nějakého řešení a mají ohodnocení podle nejlepšího možného (teoreticky dosáhnutelného – bez ohledu na omezující podmínky) řešení z daného stavu. Pro průchod do dalšího uzlu si vybíráme vždy podle nejlepšího ohodnocení daného uzlu. Uzly, které nevyhovují omezující podmínce, dále nerozvíjíme (ořezáváme celé větve).

3.6.3 Zametací technika – Plane sweep

Hlavně pro geometrické úlohy.
Myšlenka:
Prostorem P na němž probíhá řešení, proložíme zametací nadrovinu (v 2E je to přímka, v 3E rozina), tuto nadrozvinu posouváme mezi krajními body P. Nadrovina rozděluje P na levý (v něm již byla úloha vyřešena) a pravý (v něm se teprv bude hledat) poloprostor. Spolu se zametací nadrovinou udržujeme strukturu S, která obsahuje všechny informace o stavu řešení úlohy v levém polop. (řešení v levém polop. je relevantní pro řešení v pravém polop). Stav struktury S se tedy mění s polohou nadroviny. Nadrovina se posouvá do pozic, kde se stavy řešení mění. Tyto pozice se nazývají postupový plán, který se může s polohou zametací nadroviny měnit.



obr. 8.3 Nalezení průsečíku 3 přímek zametací technikou

Řešení zahájíme pro polohu zametací přímky x_{min}. Pro tuto polohu vytvoříme počáteční y-strukturu. Každé dvě sousední přímky v y-struktuře se mohou protínat. Pokud jejich průsečíky leží vpravo od zametací přímky, zařadíme je do x-struktury. V každé poloze zametací přímky pak postupujeme takto:

1. Opakující se bod v y-struktuře je průsečík dvou přímek nebo více přímek a generuje jeden nebo více výstupů.
2. Pro opakující se body zaměníme pořadí odpovídajících přímek v y-struktuře. Zaměněné přímky se mohou se svými novými sousedy protnout pouze vpravo od zametací přímky, neboť vlevo již byly všechny průsečíky nalezeny. Tyto průsečíky zařadíme do x-struktury.

Popsaný algoritmus nalezení 3 průsečíků zametací technikou má složitost řádově $O(n^2 \cdot \log_2 n)$, neboť zametací přímka bude mít v nejhorším případě $n \cdot (n-1)/2$ poloh a složitost kroků 1 a 2 v každé poloze zametací přímky je dána hledáním ve vyváženém binárním stromu y-struktury. Inicializační fáze má operační složitost řádově $O(n.\log_2 n)$, takže celkovou asymptotickou složitost neovlivní.

3.7 Princip a vlastnosti dynamického programování

Postupujeme zdola nahoru (od dílčích úloh), tabelujeme výsledky (uchováváme) dílčích úloh a používáme je – nedochází k opakovaným výpočtům, paměťově náročné.
1. Najdeme a uložíme řešení dílčích úloh,
2. Kombinací těchto získáme celé řešení
Např. věci v batohu:
cena(i+1, w)
- cena (i, w) – bez předmětu i
- cena (i, w-wi) – s předmětem i
Hodnoty jsou tabelované podle i a w.

Dynamické programování je čistá dekompozice. Dekomponované instance se dají charakterizovat malým počtem hodnot a řešení dekomponovaných instancí lze těmito hodnotami indexovat. Zároveň se dají dekomponované instance rozdělit do disjunktních tříd (stupňů) tak, že k výpočtu instancí jednoho stupně je třeba jen instancí právě jednoho jiného stupně.

Dynamické programování lze formulovat dvěma způsoby. *Rekurzivní* způsob vyjde ze zadané instance, stanoví, které podinstance je třeba řešit, až dosáhne triviálních instancí. V praxi se tento způsob nepoužívá.

Dopředná formulace také vyjde ze zadané instance, spočítá všechny složitější podinstance, až dosáhne zadané instance.

Řešení jednotlivých podinstancí se samozřejmě v obou případech zaznamenávají a jsou tak vždy počítány nejvýše jednou.

3.8 Princip a kostra – simulované ochlazování

Vyváznutí z lok.extrémů: Jsem ochoten přijmout i horší řešení, ale jen s nějakou pravděpodobností (ta záleží na velikosti zhoršení a aktuální teploty).

$\delta \to 0$	$p_{new} \to 1$	nepatrné zhoršení se přijme vždy
$\delta \to \infty$	$p_{new} \to 0$	velké zhoršení se přijme zřídka
$T \to 0$	$p_{new} \to 0$	při nízké teplotě se zhoršení přijmou s malou pravděpodobností
$T \to \infty$	$p_{new} \to 1$	při vysoké teplotě se přijmou i velká zhoršení

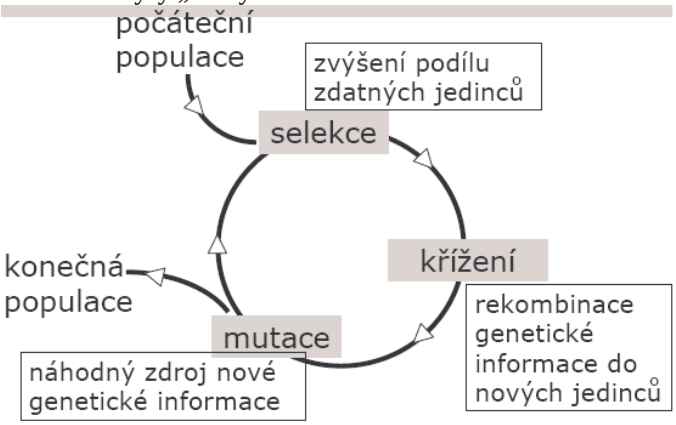
```
t=t0
repeat
  repeat
    state=try(state, t)
    if (state.better(best))
      best = state;
  until rovnováha();
  t=ochlad(t);
until zmrazeno();
```

t_0 – počáteční teplota. Na počátku velká. Buďto náhodně nebo určeno jinou heuristikou.
try – Vyberu náhodně souseda, pokud je lepší než state, beru ho. Když je horší, beru s pravděpodobností $e^{-\delta/T}$, kde δ je velikost zhoršení (new.cost() – state.cost()) a T je aktuální teplota.
Rovnováha() – pevný počet iterací nebo počet iterací je funkcí teploty
Ochlazení – nelineární t=0,9.t nebo (Hajek) $t_k = 1 / \log(1+k)$, kde k je číslo kroku a c je hloubka lokálního minima
Podle počtu přijatých změn, lze i ohřívat (ven z lok. minima)
Zmrazeno() – pevná mez teploty nebo počet nových přijatých stavů je malý,
Hikův teorém (známe hloubku lok. minima, pst vyvedení<ε)

Modifikace: - jednodušší vztah pro pst (lehčí výpočet) $p=1-\delta/T$
- pro malé t prohlížím sousedy systematicky ne náhodně (moc odmítnutých kandidátů)
- paralelizace

3.9 Princip a kostra – genetické algoritmy

Beru více stavů najednou – pravděpodobnost, že uváznu v lok.extr. je menší.
Zachovává rysy „dobrých“ řešení – funkce fitness. Genetické operace křížení a mutace.



Iterace:

```
repeat G=new_generation(G) until hotovo();
```

Překrytí generací:
- žádné – nahradí se celá G potomky a mutanty
- částečné – vyhodím horší část (např. půlku) generace a doplním potomky na pův. velikost
- úplné – nahradím jednoho (2) jedince novými potomky

Funkce fitness_select():
- přímo optimalizační kritérium (moc rychle konverguje)
- ranking – jedince beru s pstí závesející na jeho fitness
- scaling – „změkčím“ optimalizační kritérium g(x)= a.f(x) + b; zachovám střední hodnotu! (jedince seřadím dle fitness a udělám si novou linearní funkci podle fitness jedince a výběr provádím na upravené funkci)

Modifikace – messy GA – kříží se různá lokální minima získaná konstruktivní metodou; složitější křížení na základě vlastností
dobrých jedinců

3.10 Princip a kostra – tabu prohledávání

tabu je historie dané délky;
ovlivňuje okolí stavu, příp. heuristickou fci;
na modif. okolí zpravidla best-first

tabu – kritérium odmítání operací (nových stavů), např. historie inverzních ops, již prohledané stavy, četnost nějaké operace

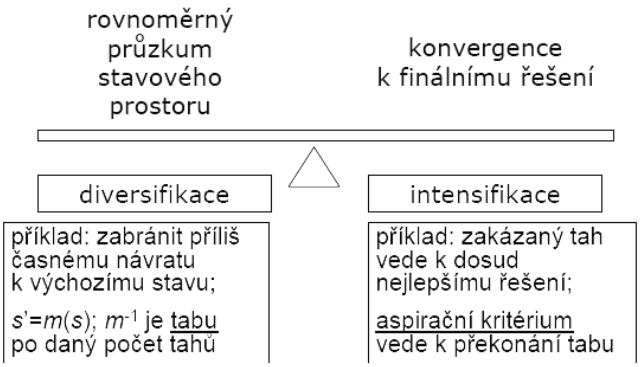
aspirační fce – nový stav beru, i když je tabu; nejlepší výsledek, nejlepší lok. výsledek, všechny jsou tabu, tak tahle
- nebo na základě sledování atributů stavu, ovlivňují aspir. fci nebo optim. kritérium (intenzifikace)

tabu-lhůta – staticky, dynamicky podle atributů
- mohu prohledávat i stavy, co nejsou řešení (s nějakou penailzací)

Paměť:
základ adaptace a učení: jaké prvky (rysy) řešení mají podstatný vliv na
- kvalitu řešení
- strukturu řešení

špatné strategické rozhodnutí může přinést více informace než dobré náhodné rozhodnutí (Glover, Laguna)

Cíle řízení



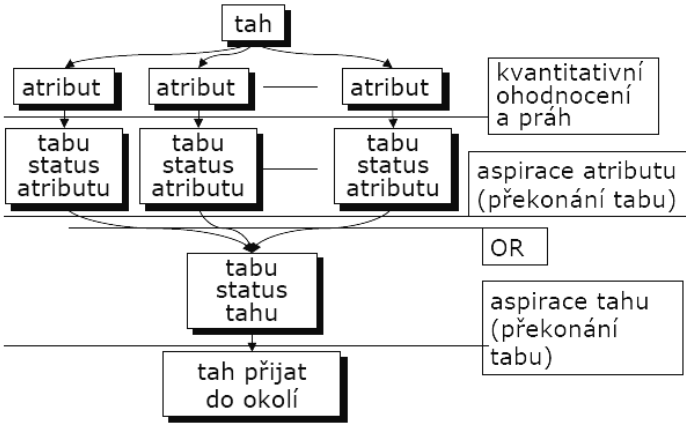
Intenzifikace a diverzifikace

- obě metody (řízení okolí a modifikace heuristické funkce) mohou sloužit oběma cílům
- řízení okolí: zpravidla vyloučení stavů, které nechceme prohledávat
  - které jsou příliš podobné známým stavům
  - které nejsou dost podobné známým stavům
- modifikace heuristické funkce:
  - pokutování příliš podobných / odměňování nepodobných
  - odměňování podobných / pokutování nepodobných

Práce s pamětí

- úplný záznam všech tahů není možný
- často potřebné zakázat tahy s podobnými vlastnostmi
- vyhledávání musí být rychlé
- ⇒ abstrakce
  - tah nebo skupinu tahů popsat pomocí atributů
  - skladovat omezený počet elitních řešení resp. jejich sousedů (explicitní paměť)
  - skladovat historii po omezený počet tahů (implicitní krátkodobá paměť)
  - skladovat omezené statistiky pro celý průběh (implicitní dlouhodobá paměť)

Řízení okolí



Volba tabu lhůty

- statická: t=7, t=sqrt(n)
- dynamická: t=5..9, t=(0,5..2).sqrt(n)
- strategie změny?

Aspirační kritéria

- základní možnost: jestliže všechny tahy jsou tabu, přijmout „nejméně tabu" tah
- aspirace tahu:
  - vede k dosud nejlepšímu řešení
  - vede k odlišnému řešení