NI-KOP (přednášky)

Ondřej Wrzecionko

ZS 2022/2023

Obsah

1 Úvod

1.1 Cíl předmětu

Heuristická optimalizace: Občas se dostaneme do situací, kdy neumíme formálně popsat, co chceme, ani neumíme napsat vyhovující algoritmus *(nedokážeme formálně popsat "chodce")*, často v souvislosti se strojovým učením.

V NI-KOP se naučíme, jak kombinatorické úlohy takto efektivně řešit. Algoritmus je většinou snadný, zkoumáme nasazení implementovaného algoritmu, experimentálně nastavujeme jeho parametry a vyhodnocujeme užitečnost.

Často nasazení provádíme jednorázově, rozumíme danému algoritmu a dokážeme ji ladit libovolně až do požadovaného výsledku. Někdy se ale software šíří mimo dosah autora, obsluha algoritmu nerozumí a program se s tím musí vypořádat sám.

Naučíme se porozumět principům a práci heuristik, složitosti řešených úloh a metodám, jak experimentovat.

→ Exaktní řešení bude náročné vždy, kontrola korektnosti daného řešení ale není obtížná, pokud nám tedy bude stačit alespoň trochu dobré řešení, máme naději na úspěch.

Profesionální řešení: nasazení heuristiky, vyladění na menší sadě úloh, statistické vyhodnocení set až tisíc běhů podle charakteru heuristiky a úloh

1.2 Hodnocení

- teoretický test v 6. týdnu semestru (30 bodů, min. 10b)
- 1. domácí úloha (experimentální vyhodnocení – 15 bodů, povinné)
- 2. domácí úloha (nasazení heuristiky – povinná)
- zkouškový test (25 bodů, min. 10b)

Při úlohách se hodnotí praktické provedení, pracovní postup, ani ne naprogramovaný kód nebo přesné rozdělení, ale především to, že přístup je racionální a výsledky jsou relevantní.

Pokud se student odchýlí od zadání, úmysl je třeba prezentovat cvičícímu, modifikovaná úloha musí splnit původní cíle a srozumitelnost zprávy musí zůstat stejná.

Programovat se musí v imperativním jazyce (libovolném), měří se veličiny, které charakterizují algoritmus, na efektivitě kódu nezáleží. Kód by měl být správný a korektní, aby se hodnotící nemusel vrtat v kódu.

1.3 Pojmy

Matematická optimalizace

Zajímá se o konečné a diskrétní problémy s konečným počtem proměnných a konečným počtem hodnot každé proměnné. Vše lze tedy vyřešit hrubou silou, ale není prakticky použitelná *(všech možných kombinací může být hodně)*.

Problém vs instance

Problém (vstupní/výstupní/konfigurační proměnné, omezení, optimalizační kritéria) vs instance (konkrétní **hodnoty**).

Problémem může být nalezení optimální cesty pro vrtačku na dané desce, instancí je pak konkrétní deska.

Konfigurační proměnné jsou to, co nastavuje hrubá síla, všechny kombinace, to, kde se dají zkontrolovat omezení a z čeho lze vypočíst hodnotu optimalizačního kritéria. S výstupními proměnnými se shodují jen u některých problémů.

1.4 Problémy

Problém jsem schopný zakódovat (nejjednodušeji binárně), pak každá instance je charakterizována řetězcem 0 a 1, problém charakterizuji instancemi s výstupem ano, je to tedy jazyk (podmnožina {0, 1}*)

Problémy bez optimalizačního kritéria

Nechť I je instance problému, Y konfigurace, $R(I, Y)$ omezení *(Y je řešením)*:

- rozhodovací problém:
  - Existuje Y takové, že $R(I, Y)$?
  - Platí pro všechna Y, že $R(I, Y)$?
- konstruktivní problém: sestrojit nějaké Y takové, že $R(I, Y)$
- enumerační problém: sestrojit všechna Y taková, že $R(I, Y)$
- početní problém: kolik existuje Y takových, že $R(I, Y)$

Problémy optimalizační

Nechť I je instance problému, Y konfigurace, $R(I, Y)$ omezení a $C(Y)$ optimalizační kritérium *(cenová funkce)*:

- optimalizační rozhodovací problém: existuje Y takové, že $R(I, Y)$ a $C(Y)$ je alespoň tak dobré, jako daná konstanta K?
- optimalizační konstruktivní problém: sestrojit nějaké Y takové, že $R(I, Y)$ a $C(Y)$ je nejlepší možné

- optimalizační enumerační problém: sestrojit všechna Y taková, že $R(I, Y)$ a $C(Y)$ je nejlepší možné
- optimalizační početní problém: kolik existuje Y takových, že $R(I, Y)$ a $C(Y)$ je alespoň tak dobré, jako daná konstanta K?
- optimalizační evaluační problém: zjistit nejlepší možné $C(Y)$ takové, že $R(I, Y)$

Problém batohu

Je daná množina U, pro každý prvek

$$u \in U$$

je dáno

$$c(u), w(u) \in \mathbb{N}$$

Zkonstruujte množinu U' takovou, aby byl součet vah w(u) menší než M a součet cen c(u) co největší.

Vstupní proměnné	n, M, w(u), c(u)
Konfigurační proměnné	$U' \subseteq U$
Omezení	hmotnost batohu
Optimalizační kritérium	maximální cena

Hledání konfiguračních proměnných je nalezení takové sestavy věcí (konstruktivní problém) v batohu, aby nebyl přetížen (tvrdá podmínka) a cena věcí byla maximální (měřítko).

Konfigurace	ohodnocení konfiguračních proměnných
Řešení	konfigurace, která vyhovuje omezujícím podmínkám
Optimální řešení	má nejlepší hodnotu optimalizačního kritéria
Suboptimální řešení	má přijatelnou hodnotu optimalizačního kritéria

Existuje mnoho variant problému batohu – existuje plnění s cenou K? Nalézt plnění s nejlepší cenou. Nalézt cenu nejlepšího plnění. Všechny tyto varianty mají stejné vstupní/konfigurační proměnné, omezení, ale výstup je jiný. Proto konfigurační proměnné potřebujeme.

Problém splnitelnosti formule

Dána množina n proměnných, Booleova formule v konjunktivní normální formě. Cílem je nalézt, zda je formule splnitelná, výstupem je ano/ne.

2 Experimentální práce s algoritmy

2.1 Účely

Teorie: srovnání se známými algoritmy, s optimem.
Aplikace: vhodnost algoritmu pro zamýšlenou funkci.

Složitosti:

- nejhorší případ (vzácný a nezajímavý, složitá analýza)
- průměrný případ (proveditelné pro jednoduché případy)

→ experimentální vyhodnocení = složitost jednotlivých kroků nelze spočítat.

Postup

Otázka co chci zjistit
→ plán experimentu
→ provedení experimentu
→ interpretace výsledků
→ odpověď *(toto běželo x hodin, proto…)*

2.2 Metriky

Zajímají nás metriky vstupu a výstupu a jejich závislost.

Algoritmus: heuristika => problémově závislé části algoritmu => parametry algoritmu
→ v souvislosti s algoritmem se zkoumá počet navštívených konfigurací

Implementace: datové struktury – kódování – platforma
→ v souvislosti s implementací se zkoumá doba běhu (čas CPU)

Při měření závislostí metrik někde můžeme držet stejné metriky vstupu, u některých to ale nejde (únava topiče).

U SATu se zjistilo, že vstupní metrikou je poměr počtu klauzulí k počtu proměnných.

Při generování sady instancí musím pro každou instanci se **zadanou metrikou** nechat **stejnou** pravděpodobnost. Pokud instance sbírám, snažím se získat co nejvíce instancí (se stejnou metrikou).

Toto proženu algoritmem, změřím data, proženu to statistikou, abych se zbavil variance a interpretuji.

2.3 Statistika pro 1 hodnotu

Nejlepší je použít průměr nebo medián, použít vhodné parametry rozložení (μ, σ) a následně kvalitně reprezentovat.

2.4 Srovnání algoritmů

Počet kroků algoritmu nechť je náhodná proměnná, pokud má A lepší parametry, je lepší. Pokud pro každou instanci A je lepší, pak A dominuje.

2.5 Randomizovaný algoritmus

Všechny hodnoty generátoru náhodných čísel jsou stejně pravděpodobné + potlačení variance z randomizace.

Spolehlivá data můžeme ověřit pravděpodobnostním mechanismem testování hypotéz.

Ale...

Někdy nejsme schopni charakterizovat problém → nemáme věrohodný generátor a nemůžeme potlačit varianci. Pak můžeme použít standardní sady problémů k určité úloze, kde najdeme i možnost porovnání s již existujícími algoritmy. Tomuto se říká **inženýrská algoritmika** – srovnávat lze pouze vzhledem k dané sadě instancí.

2.6 Interpretace experimentu

Data: chování na konkrétních instancích s konkrétními parametry (pohled zvenčí, mnoho jednotlivých dat)

Interpretace: obecný závěr – musíme provést extrapolaci, bývá kvalitativní, pohled zevnitř, jednoduchá formulace

2.7 Prezentace experimentu

Při prezentaci čtenáře zajímá důležitost otázky, relevance experimentu, reprodukovatelnost jeho důvěra v interpretaci a přijetí odpovědi.

Autor tedy musí znát cílovou skupinu a předat ji data, získat její důvěru a přesvědčit ji. K tomu slouží tzv. IMRaD.

2.8 IMRaD

IMRaD se skládá z:

- Introduction: je potřeba X, široce zaměřené
- Methods & Results: úžeji zaměřené, obsahuje reporty
- Discussion (& Conclusion): udělali jsme X, široce zaměřené

2.8.1 Introduction

Proč byla studie provedena? Jaký byl účel výzkumu? Jaká byla otázka? Jakou hypotézu jsme testovali?

→ buduje výchozí bod pro další výklad

2.8.2 Methods

Kdy, kde a jak jsme experimenty provedli? Proč jsme použili dané metody? Jak jsme ověřili korektnost? Jaký jsme použili experimentální materiál a proč?

2.8.3 Results

K jakým jsme došli výsledkům? Jakou jsme získali odpověď? Potvrdili jsme hypotézu?

→ používáme zde vždy **pouze** všechna relevantní data

2.8.4 Discussion

Co odpověď může znamenat? Jaký má vztah k dosavadnímu výzkumu? Jaké má perspektivy?

→ vše, co podporuje i nepodporuje tvrzení článku, ale ne zbytečnosti

2.8.5 Zbytečnosti

Tabulky a grafy musí být vždy dohromady, jedno bez druhého nefunguje dobře. Chybějící sloupce v tabulkách jsou také na škodu. Grafy musí obsahovat srozumitelné sdělení. Popisky musí být výstižné. Pozor na pravdivosti tvrzení!

2.9 Pro KOPy

Cílovou skupinu známe, zadání také. Důležité je předat důvěru a přesvědčení ve výsledky + že autor rozumí tomu, co dělá.

3 P a NP

3.1 Složitost

Výpočetní složitost: velikost instance určuje čas výpočtu
Paměťová složitost: velikost instance určuje spotřebu paměti

Jak můžeme měřit velikost instance? Buď **hrubou** mírou *(počet prvků instance – věcí v batohu, uzlů v grafu …)* nebo **jemnou** mírou *(počet bitů nutných k zakódování instance).*

Jak měřit čas výpočtu? Buď počtem **typických operací**, nebo počtem **kroků** jednotného výpočetního modelu.

3.2 Výpočetní modely

Turingův stroj, RAM stroj (adresace), Booleův obvod (hradla)

Turingův stroj

Program M deterministický Turingův stroj řeší **rozhodovací problém** v čase t / s pamětí m, jestliže se výpočet zastaví po t krocích / využije nejvýše m paměťových buněk pro každou instanci.

Instanci kódujeme do řetězce $\{0,1\}^*$, **způsob kódování** instance **neovlivní** čas výpočtu více než **polynomiálně**.

3.3 Třída P

Rozhodovací problém **patří do třídy P**, pokud pro něj existuje program pro deterministický Turingův stroj, který jej řeší v čase $O(n^k)$, kde n je velikost instance a k je konečné číslo.

(Note: Existuje program, ne, že ten algoritmus známe.)

3.4 Třída PSPACE

Rozhodovací problém **patří do třídy PSPACE**, pokud pro něj existuje program pro deterministický Turingův stroj, který jej řeší v paměti $O(n^k)$, kde n je velikost instance a k je konečné číslo.

3.5 Třída EXPTIME

Rozhodovací problém **patří do třídy EXPTIME**, pokud pro něj existuje program pro deterministický Turingův stroj, který jej řeší v čase $O(2^{P(n)})$, kde $P(n)$ je polynom ve velikosti instance n. Platí, že PSPACE $\subset$ EXPTIME.

3.6 Třída NP

Motivace

NP = Nedeterministicky polynomiální. Problémy, které **nemusí** mít polynomiální algoritmus, ale kde **cesta stavovým (konfiguračním) prostorem** je vždy **únosně dlouhá** (polynomiální ve velikosti instance).

Řešení problému

Pro každý problém označíme Π_{ANO} množinu instancí, které mají výstup ANO, Π_{NE} množinu instancí, které mají výstup NE.

Program M pro nedeterministický Turingův stroj řeší rozhodovací problém Π v čase t, pokud se výpočet zastaví po t krocích pro každou instanci $I \in \Pi_{ANO}$ problému Π. *(Nic neříkáme o instancích* Π_{NE}

Třída NP

Rozhodovací problém **patří do třídy NP**, pokud pro něj existuje program pro **nedeterministický** Turingův stroj, který každou instanci $I \in \Pi_{ANO}$ řeší v čase $O(n^k)$, kde n je délka vstupních dat a k konečné číslo.

Rozhodovací problém **patří do třídy NP**, právě když pro každou instanci $I \in \Pi_{ANO}$ existuje konfigurace Y taková, že kontrola, zda Y je řešením, patří do P. *(omezující podmínky lze vyhodnotit v polynomiálním čase, Y nazýváme certifikátem nebo svědkem)*

P = NP ?

Určitě víme, že P $\subseteq$ NP. Jestli platí P = NP? Kdo ví …

"Otočení" NP problému

Je graf bez Hamiltonových kružnic? Je Booleova formule nesplnitelná? NP zde nefunguje, jak najdeme certifikát? Přecházíme z NP problému $\exists Y, R(I, Y)$ k coNP problému $\forall Y, \neg R(I, Y)$.

Svědkové

NP problém: Y je krátký svědek odpovědi ANO, krátký svědek odpovědi NE neexistuje, ale jsme schopni udělat krátké vyhodnocení každé konfigurace.

3.6.1 Třída P, NP, co-NP

P patří do NP $\cap$ coNP *(jsme schopni najít svědka ANO i NE, je to řešení problému samotného).* Jsou ale problémy, které nejsou P, a jsou zároveň NP i co-NP *(existuje prvočinitel celého čísla N, jehož poslední číslice je 7?)*

3.7 Vztahy

P = co-P, P $\subseteq$ NP $\cap$ co-NP *(máme certifikát $\exists$ i $\forall$)*

Pokud by platilo P = NP, pak by platilo NP = co-NP.

3.8 Horší než NP

QSAT$_2$: Booleovská formule $F(X_1, X_2)2n$ proměnných. $\exists Y_1, \forall Y_2 F(Y_1, Y_2) = 1$? Certifikátem je Y_1, jak ale zkontrolovat druhou část, když je problém kontroly v co-NP? Tyhle problémy můžeme neustále zhoršovat, až po QSAT$_k$ ...

Takto konstruktivně definuji třídy problémů Σ_i^P, pro které existuje krátký $\exists$-svědek a problém jeho kontroly je třídy Π_{i-1}^P a také třídu problémů Π_i^P, pro které existuje krátký $\forall$-svědek a problém jejich kontroly je třídy Σ_{i-1}^P.

Například tedy Σ_0^P = P = Π_0^P, Σ_1^P = NP, Π_1^P = co-NP. Jak je to dál nevíme ...

4 NP-těžké (NPH) a NP-úplné (NPC) problémy

Problém Π je **X-těžký**, jestliže se efektivní řešení všech problémů z třídy X dá **zredukovat** na efektivní řešení problému Π.

Problém Π je **X-úplný**, jestliže je X-těžký a sám patří do třídy X.

Efektivní řešení = v polynomiálním čase s omezenou chybou

Zredukovat na řešení X = **vyřešit** pomocí nějakého řešiče pro X.

Nejtěžší problém = ten, **na** který lze převést všechny ostatní

(Pokud jde Π_1 *převést na* Π_2*, pak je* Π_1 *nejvýše tak těžký, jako* Π_2 *a* Π_2 *je nejméně tak těžký jako* Π_1

Rozhodovací problém Π_1 je **Karp-redukovatelný** na Π_2, pokud existuje polynomiální program pro (ne)deterministický Turingův stroj, který převede každou instanci I_1 problému Π_1 na instanci I_2 problému Π_2

Pokud dokážu zredukovat Π_1 na Π_2 a Π_2 na Π_1, jsou **polynomiálně ekvivalentní** + tato relace je ekvivalentní.

Problém Π je **NP-těžký**, jestliže pro všechny problémy z NP, jsem schopný je převést na Π *(ale já neumím převádět mimo NP)*. U **NP-úplný**ch problémů to jsem již schopen popsat. Problémy v **NPC** jsou **nejtěžší** problémy v NP.

Všechny NPC problémy tvoří třídu ekvivalence.

4.1 Cookova věta

SAT je NP-úplný, tedy NPC není prázdná a známe tisíce NPC problémů. Pokud bychom našli efektivní algoritmus na jeden, pak by existoval na všechny, ale nevypadá to tak. NPC = NPH ∩ NP

Důsledky: pokud je problém v NP, pak není horší než NP a existující certifikát / svědek jsme schopni efektivně zkontrolovat a prakticky ho převést na známé problémy s cca efektivními řešiči (SAT). Exaktní řešení NP problémů ale můžeme prakticky vyloučit.

4.2 Třída NPO

Optimalizační problém patří do NPO, pokud je výstup, omezující podmínky i optimalizační kritérium vyhodnotitelné v P. Patří do PO, pokud je NPO a existuje řešení v polynomiálním čase.

Pokud nelze vypočíst optimalizační kritérium v P, není to ani NPO, je to těžší, bad news.

4.3 NPI

Problémy, které nemohou mít polynomiální algoritmus ani na ně nemůže být převeden SAT (např. izomorfismus grafů)

5 Optimalizační problémy

5.1 Pseudopolynomiální algoritmy

Tváří se jako P, ale nejsou polynomiální. Například problém batohu se tváří jako polynomiální $O(n \cdot M)$, ale ve skutečnosti M nesouvisí s velikostí instance.

Definice: Algoritmus, jehož počet kroků závisí polynomiálně na velikosti instance, ale závisí dále na vstupní proměnné, který s velikostí instance nesouvisí.

5.2 Aproximativní algoritmy

U problému batohu lze věci vkládat seřazené podle klesajícího poměru cena / hmotnost, vkládá se věci, dokud není překročena nosnost. Tento algoritmus má **polynomiální** složitost, a lze dokázat, že výsledné řešení má cenu $\geq 50\%$ optimálního řešení.

U těchto programů jsme schopni spočítat **relativní kvalitu / chybu** jako maximum z hodnot aproximativních / optimálních kritérií …

5.2.1 Třída APX

Algoritmus APR pro problém Π je **R-aproximativní**, jestliže každou instanci problému Π vyřeší v polynomiálním čase s relativní kvalitou R (chybou ϵ).

Optimalizační problém Π je **R-aproximativní**, jestliže pro něj existuje R-aproximativní polynomiální algoritmus. $R(\epsilon)$ nazveme **aproximativním prahem** problému Π.

Optimalizační problém Π patří do třídy **APX**, jestliže je R-aproximativní pro **konečné** R.

5.2.2 Známé aproximativní prahy

Pro uzlové pokrytí je to $\epsilon < 1/2$, pro problém batohu libovolně malé číslo (ale ne 0), pro problém obchodního cestujícího v optimalizační verzi je $\epsilon = 1$, neexistuje tedy aproximativní algoritmus.

5.3 PTAS

Algoritmus APR, který pro každé $1 > \epsilon > 0$ vyřeší každou instanci I problému Π s relativní chybou nejvýše ϵ v polynomiálním čase nazveme **polynomiální aproximační schéma problému** Π. Problém patří do třídy **PTAS**, pokud pro něj existuje polynomiální aproximační schéma.

Například problém batohu jsme takto schopni aproximovat algoritmem PTAS-KNAP (pamatuji si všechny konfigurace batohu, které obsahují $1/\epsilon$ věcí do méně a doplníme je algoritmem APR-KNAP.

5.4 FPTAS

Polynomiální aproximační schéma APR, jehož čas výpočtu závisí polynomiálně na $1/\epsilon$, nazýváme **plně polynomiální aproximační schéma**. Problém Π patří do třídy **FPTAS**, pokud pro něj existuje plně polynomiální aproximační schéma.

5.5 APX redukce

Nechť $\Pi_1, \Pi_2 \in NPO$. Pro libovolné $r > 1$ existují dvě funkce tak, že: $f(I_1, r)$ zachovává existenci řešení a mění instanci I_1 na instanci I_2, to jsem schopný aproximativním algoritmem převést na Π_2 s relativní kvalitou r, pak i tuto instanci jsem schopný převést $g(I_1, y, r)$ zpět na řešení y instance I_1 problému Π_1 …

APX redukce se značí $\Pi_1 \propto^{APX} \Pi_2$. Pokud něco převedu na APX nebo PTAS problém Π_2, pak i Π_1 patří do APX nebo PTAS.

5.6 Těžké třídy

Problém Π je **NPO-těžký**, jestliže $\forall \Pi^x \in$ NPO, $\ddot{}\Pi^x \propto^{APX} \Pi$, NPO-úplný, pokud je NPO-těžký a zároveň je v NPO.

Problém Π je **APX-těžký**, jestliže $\forall Pi^x \in$ APX, $\Pi^x \propto^{APX} \Pi$, APX-úplný, pokud je APX-těžký a zároveň je v APX.

6 Stavový prostor

U **dynamického** programování zakládám řešení vždy na **optimálním** řešení problému nižšího stupně.

Batoh: Platí, že následník každé konfigurace má větší celkovou váhu a cenu, následník **nepřípustné** konfigurace je nepřípustná konfigurace.

6.1 Formální definice

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** konfiguračních a vnitřních proměnných je **stav** algoritmu A řešícího I.

Nechť $S = s_i$ je množina všech stavů algoritmu A řešícího I a $Q = q_j$ množina **operátorů** $S \to S$ takových, že $q_j(s_i) \neq s_i$ pro všechny kombinace (vždycky to nějak změní). Pak dvojici (S,Q) nazveme **stavovým prostorem** algoritmu A řešícího I.

Akce: aplikace jednoho konkrétního operátoru q_j na jeden konkrétní stav s_i. Stavový prostor se dá namapovat na **orientovaný graf** $H = (S, E)$, kde hrana (s_i, s_j) odpovídá jedné akci, vrcholy jsou jednotlivé stavy.

Okolí stavu je množina stavů, do kterých se dostanu **jedním** krokem, aplikací jedné operace. Stavy z okolí se nazývají **sousední** stavy.

k-okolí stavu s je množina stavů, dosažitelných s aplikací nejméně jedné a nejvýše **k** operací/kroků. Výměny uvnitř grafu nejsou užitečné sama o sobě. Pouze inverzní operátory taky nestačí.

6.2 Grafy stavového prostoru

- **acyklický**: omezená délka cesty, jednoduché řízení heuristiky, *hladové algoritmy*
- **cyklický**: vyžaduje komplikovanější řízení, pokročilé heuristiky

Pro každé dva stavy nazveme délku nejkratší cesty z s_1 do s_2 v grafu H **vzdáleností** uzlu s_2 od s_1. Aby byla heuristika **pokročilá**, musí platit:

- **dostupnost**: mezi každými dvěma uzly musí být cesta → graf musí být silně souvislý
- **symetrie**: pro každé dva stavy musí být jejich vzdálenost s_1 do s_2 a s_2 do s_1 přibližně **stejná**

Konfigurace je obecně **podgraf** grafu G v závislosti na algoritmu. **Uzel stavového grafu** je ten daný stav, hranou je operace, pozor na příklad v úkolech souvisejících s grafy.

POZOR !!! Některé úlohy mají za úkol najít **cestu** ke stavu jiného objektu *(hra Sokoban)*. Množina stavů je pak **množina posloupností akcí**, operace nad stavem **přidej na konec**. Něco jiného je pak **stav scény** a **stav algoritmu**.

6.3 Strategie pohybu stavovým prostorem:

6.3.1 Úplná strategie

Úplná strategie navštíví **všechny** stavy, kromě těch, o kterých víme, že nedávají optimální řešení *(otevírá cestu k prořezávání)*

Úplný algoritmus dokáže odpovědět, že instance nemá řešení. Použití úplné strategie v lokální heuristice dává úplný algoritmus *(za cenu složitosti)*.

6.3.2 Systematická strategie

Systematická strategie je úplná strategie, která navštíví každý stav **nejvýše jednou**. Typicky systematické strategie obsahují proměnné:

- **open**: neprozkoumaní sousedé *(fronta – BFS, zásobník – DFS, prioritní fronta – best first)*
- **closed**: prozkoumané stavy
- **state**: aktuální stav
- **best**: nejlepší nalezený stav

Pokud neuvažujeme **prořezávání**, je nejhorší případ roven **hrubé síle** a tento případ nastane v případě **neexistujícího** řešení. Tyto strategie naleznou *(optimální)* řešení, existuje-li.

6.4 Lokální heuristiky

U těchto heuristik platí, že struktura open má **jen jednu** položku, neprozkoumám typicky všechny možné stavy.

6.4.1 Best only

Vrátím **nejlepší** z okolních řešení. Pokud jsou všechna okolní řešení horší, vrátím prázdný stav. **Ne**záleží na pořadí procházení sousedů.

6.4.2 First improvement

Najdu prvního souseda, který je lepší, pokud žádný není, vrátím prázdný stav. **Záleží** na pořadí procházení sousedů → je nutná randomizace.

6.5 Závěrem

Většinou provádíme **kratší**, jednoduché, ale **rychlé** akce a pokud se to po nějakých MAX_FLIPS nepovede, provedeme **složitější** a radikální akci.

7 Randomizované algoritmy

Randomizovaný algoritmus: založen na náhodné volbě, jeho vlastnosti jsou tedy vyjádřeny statisticky.

Monte Carlo algoritmy: dosažený výsledek je náhodná proměnná, čas běhu je pevný pro danou instanci *(budu gamblit do půlnoci, otázkou je, kolik prohraju)*

Las Vegas algoritmy: dosažený výsledek je vždy správný, čas běhu je náhodná proměnná *(všechno prohraji, otázkou je za jak dlouho)*

Například u randomizovaného SATu mám počáteční ohodnocení se stejnou pravděpodobností, a pak postupně opakuji algoritmus a získávám tím lepší výsledek = lokální heuristika s **náhodnou** volbou. *(Monte Carlo)*

Příkladem může být i Rabin-Millerův test prvočíselnosti (pravděpodobnost, že číslo je složené, je $1 - \frac{1}{4}^r$ *(Monte Carlo)*) nebo Quicksort *(Las Vegas)*.

Výhody: strukturní **jednoduchost**, očekávaná kvalita může být lepší než zaručená kvalita aproximativních algoritmů, u Monte Carlo se dá nezávislým opakováním **zlepšit** kvalita.

Kombinace s deterministickými prvky tedy poskytuje **nestrannost** (náhodný start, náhodně vybraný sousední stav, náhodně vybraný krok z množiny, kde heuristická funkce dává stejnou hodnotu).

Formální analýza je **složitější**, většinou očekáváme střední hodnotu. Proto si vybíráme **experimentální** charakterizaci – primární metrikou je počet kroků, úspěch (na **jedné** instanci: rozdělení počtu kroků, (korigovaná) distribuční funkce, na **více** instancích:)

8 Nasazení heuristik

8.1 Instalační závislosti

Máme problém řešení instalačních závislostí (balíčky jsou v různých repozitářích, mají na sobě různé závislosti, nebo konflikty).

Existují **distribuce**, které tento problém převádí na **SAT**. Instalací balíčku se rozumí daný balíček, požadavek na instalaci balíčku je klauzule (a), **závislost** je $(\neg a + b)$, konflikt $(\neg a + \neg b)$ a hledám **ohodnocení** proměnných (= rozhodnutí o instalaci) takové, aby byla hodnota **1** (nainstaloval jsem vše), optimalizačním kritériem může být **velikost** stažení, aktuálnost systému.

8.2 Práce s heuristikami

Každý algoritmus / heuristika má nějaké číselné **parametry**, které jsou nesrozumitelné koncovému uživateli a navzájem se ovlivňují.

Buď tedy uživatel může zkusit **postupně** zjišťovat, kde to funguje optimálně, nebo **pochopí**, co který parametr dělá, sleduje charakter a využije to.

white-box: pracujeme s omezenou sadou instancí, provádíme detailní měření, abychom všemu porozuměli, modifikace heuristiky

black-box: plná sada instancí, měření výsledků, ověření kvality a výkonu bez modifikace heuristiky

Parametry obecně nejsou nezávislé, někde je závislost známa, někde je nutná ověřit, v každém případě se vždy jedná o **konfiguraci heuristiky**.

8.2.1 Faktorový návrh

Vyzkouším **všechny** kombinace hodnot parametrů → pokud neznám chování algoritmu. Nevýhodou je ale **více** parametrů, musíme odhadnout **rozsah** a **krok** každého parametru.

8.2.2 Cesta prostorem parametrů

Snažím se **postupnými změnami** parametrů dostat k ideální variantě. Zabere méně práce, času, musíme ale **porozumět** algoritmu a nevíme, jestli to bude **konvergovat**.

8.3 Zpráva o nasazení

Velmi důležité je **shrnutí**: v jakém rozsahu lze heuristiku **nasadit**, jaká je **kvalita**.

White box: postup, důvod postupu, důvod nastaveného rozsahu faktorového návrhu, důvody kroků při nastavování parametrů včetně slepých uliček, popis experimentů a jejich interpretace.

Black box: prokazované tvrzení, návrh průkazného experimentu, jeho provedení a interpretace (IMRaD).

9 Metoda simulovaného ochlazování

Pokud používáme jednoduché **lokální** heuristiky, může se nám stát, že **uvázneme** v lokálním **optimu**. Při řešení tedy musíme správně vyvážit **diverzifikaci** (rovnoměrný průzkum stavového prostoru, připouštíme akci, která vede k horšímu řešení) a **intenzifikaci** (konvergence k finálnímu řešení, nepřipouštíme akci vedoucí ke zhoršení řešení).

Diverzifikace: pomocí ní můžeme zvětšit prohledávané okolí, používají ji **pokročilé** heuristiky.

9.1 Simulované ochlazování

Máme jednu konfiguraci, řídíme ji sekvenčně, využíváme diverzifikaci. Odpovídá postupnému ochlazování taveniny → můžeme ochlazovat opatrně, čímž vznikají velké krystaly, nebo prudce, čímž vzniknou malé krystaly.

Teplota ochlazování

Teplotou rozumíme řídící parametr diverzifikace *(jak často připouštíme akci vedoucí k horšímu řešení)*.

Lokální heuristiky snadno **uváznou** v lokálním minimu, vrátí prázdný stav, když neexistuje lepší soused *(first improvement)*, snadno uváznou v lokálním optimu.

Oproti tomu **pokročilé** heuristiky povolují také horší řešení s určitou pravděpodobností úměrnou zhoršení a teplotě *(u first improvement tedy vybereme náhodně stav, pokud je lepší, přijmeme ho, jinak ho přijmeme na základě zhoršení / náhody)*.

Ve vzorci $random(0,1) < exp(-\frac{\delta}{T}$ je δ zhoršení (když jde k nule, je malé, přijme se často, když se zvětšuje, přijme se méně častěji) a T teplota (když je nízká, zhoršení se přijmou s malou pravděpodobností → **intenzifikace**, při vysoké se přijmou i velká zhoršení → **diverzifikace**.

Při simulovaném ochlazování tedy postupně snižujeme teplotu: máme nějakou **počáteční teplotu**, pak funkci **frozen**, která určuje, zda se má ochlazování ukončit, **equilibrum**, které určuje, jestli je systém v "rovnovážném stavu" (pokud ano, sníží se teplota) a **cool**, která sníží teplotu.

Všechny tyto funkce dohromady určují **rozvrh ochlazování** – ten je předem určen, nebo ho řídíme zpětnou vazbou.

Stavový prostor, omezující podmínky a počáteční řešení jsou problémově závislé záležitosti lokálních iterativních heuristik.

Hodnoty cen bychom měli **normalizovat** na stejný rozsah pro všechny instance, aby nebyl problém při dělení s teplotou *(koruny vs haléře, ale stejná teplota.?)*

Ochlazování

Ochlazování: typicky cool(T) $= \alpha \cdot T$, kde $0.8 < \alpha < 0.999$

Rovnováha: typicky je určena pevným počtem kroků N. Jak cool, tak ekvilibrium spolu souvisí – měníme **délku ekvilibria** N a **koeficient chlazení** α, abychom dosáhli T_k v kroku s.

Počáteční teplota

Počáteční teplota: chceme ji nastavit tak, aby bylo přijetí zhoršující akce pravděpodobné – zjistíme to na základě hloubky lokálních optim δ, dá se vypočítat ze sady zhoršujících akcí.

Kdy to zastavit?

Metoda frozen, buď na pevné hodnotě teploty, nebo pokud četnost změn (k lepšímu) klesne pod nastavenou mez.

9.2 Stavový prostor

Simulované ochlazování bude fungovat, pokud je stavový prostor **symetrický** = pravděpodobnost generování akce a ze stavu $s_1 \to s_2$ je stejná, jako pravděpodobnost generování a^-1 ze stavu $s_2 \to s_1$, počet kroků tedy **roste** se vzdáleností uzlů.

Nějakým způsobem chceme pracovat s **omezujícími podmínkami** (můžeme využít relaxaci, opravit konfiguraci.?), **počátečním řešením** – buď zvolíme náhodné a spustíme vícenásobně, nebo zvolíme **konstruktivní** počáteční řešení.

9.3 Práce s heuristikou

Nezapomínáme na vývoj ve dvou fázích – **white box** (omezená sada instancí, detailní měření, porozumíme problému) $\to$ **black box** (plná sada instancí, měříme výsledky, ověříme kvalitu, výkon, už neměníme heuristiku).

10 Simulovaná evoluce

10.1 Principy

Oproti simulovanému ochlazování a lokálním heuristikám zde máme **více** stavů **najednou** (zpravidla konstantní počet), což nám **brání** uváznutí v jediném minimu *(uvázne třeba jeden stav, ale ne všechny)*.

Operátor: kromě unárních operátorů zde máme i **binární** operátory (SxS → S, SxS → SxS, křížení).

10.2 Analogie

biologie	**optimalizační problémy**
jedinec	konfigurace
genetická reprezentace	konfigurace
gen	proměnná kódování
alela	hodnota proměnné
generace	aktuální množina reprezentací konfigurací
mutace	unární operátor
křížení	binární operátor
zdatnost (fitness)	optimalizační kritérium
konvergence	rozšíření kvalitní konfigurace
degenerace	rozšíření konfigurace uvázlé v lokálním minimu
biodiverzita	diverzita populace

10.3 Kostra

Počáteční populace → **selekce** (zvýšení podílu zdatných jedinců) → **křížení** (kombinace do nových jedinců) → **mutace** (náhodný zdroj nové mutace) → znova selekce, nebo konečná populace.

10.4 Evoluční algoritmy

Nás se bude týkat především **genetický algoritmus** (reprezentace nuly a jedničky, dominuje křížení, malá mutace), další jsou **genetické programování** (stromy), **evoluční strategie** (vektor reálných čísel a mutace), **evoluční programování** (automat).

Společné rysy

- více stavů
- interakce stavů, kde nový stav je jejich kombinace
- prostředky diverzifikace *(vede k horšímu řešení)*: mutace
- prostředky intenzifikace *(vede k lepšímu řešení)*: selekce pro rekombinaci v další generaci

Charakteristické rysy

- reprezentace jednotlivých stavů
- unární, binární operátory
- selekce pro konstrukci následující generace
- vztah stávající a vznikající generace (náhrada / soutěž)

10.5 Generace

Obecně mám μ individuí (rodičů), které vyprodukují λ jiných individuí (potomků), přičemž μ individuí má pokračovat $\rightarrow$ jak to spojit?

Náhrada: $\lambda = \mu$, nová generace nahradí původní, nebo **náhrada s elitismem**: několik málo elitních jedinců ze staré generace zůstává nebo **Soutěž**: z $\mu + \lambda$ individuí vybereme μ nových.

Někdy mám zvláštní případy – $\lambda = \mu = 1$, pak se jedná o jednoduché heuristiky; pokud $\lambda = 1, \mu > 1$, jedná se o začlenění nového individua (**steady-state**).

U genetického algoritmu je typická **náhrada**.

11 Genetické algoritmy

11.1 Kódování

Klasická formulace je **binární řetězec**, i kdyby se jednalo o specifický problém – někde je bez problému (optimální podmnožina, SAT), někde těžší (vektor proměnných → bin packing – pro každou věc číslo kontejneru, permutace indexů měst…)

11.2 Řízení operátorů

V naší kostře evolučních algoritmů je pravděpodobnost **křížení** typicky **1** (vždy se bude křížit), oproti tomu pravděpodobnost **mutace** bude **velmi nízká** (typicky 0.03).

Křížení: náhodně zvolíme bod, kde se binární řetězce **rozdělí** a následně spojí do jednoho. **1**|000, 0|**101** -> **1|101**. Další možnost je **dvoubodové** křížení, kde se tyto body zvolí dva. **1**|00|**0**, 0|**10**|1 -> **1|10|1**. Nebo také **uniformní** křížení – vygenerujeme náhodný vektor 0 a 1, kde je 0, bereme z prvního rodiče, kde je 1, z druhého.

Aby zůstala zachována kostra klasického genetického algoritmu, zavedla se **inverze**: zpřeházím genom, ale ponechám význam proměnných (jak je to přeházené).

Selekce: má za cíl způsobit, aby početní zastoupení jedince v populaci odpovídalo jeho zdatnosti, **vyvažuje** diverzifikaci a intenzifikaci.

Selekční tlak: pravděpodobnost výběru **nejlepšího** jedince, má dva extrémy: $p = 1$ = intenzifikace, $p = 1/n$ = nezáleží na zdatnosti, diverzifikace.

Velký selekční tlak = nebezpečí **degenerace** populace (uváznutí v lokálních optimech), **malý** selekční tlak = pomalá **konvergence**, pokud šum z mutace převáží pomalou konvergenci = **divergence**. Pokud snižujeme selekční tlak, chceme snížit i **pravděpodobnost** mutace.

11.3 Ruletový výběr

Při výběru určíme pomocí pravděpodobností jednotlivých prvků v podstatě koláčový graf – **ruletu**. Provedeme m **náhodných** voleb úhlu = m prvků, jeden prvek tedy můžeme vybrat vícekrát.

Vícenásobné vybrání jednoho prvku se dá kompenzovat **univerzálním stochastickým vzorkováním** = odměříme náhodný úhel a odměříme $m - 1$ krát poměrný úhel.

Lineární škálování

Lineární škálování: naškálujeme velikost políček rulety podle **rozdílu zdatnosti** prvních dvou prvků dělený rozdílem mezi nejlepším a nejhorším: $Z = z_1 + (z - z_{min}) \cdot \frac{z_2 - z_1}{z_{max} - z_{min}}$.

Výsledný selekční tlak (pravděpodobnost výběru nejlepšího jedince) je **poměr** zdatnosti **nejlepšího** jedince a **průměru**. $c = \frac{Z_2}{Z_{avg}}$

Další

Ranking: rozhoduje pouze pořadí, **Zkrácený výběr**: vybírá pouze podle prvních prvků, **Turnajový výběr**: náhodný výběr r jedinců a z nich nejlepší, až do naplnění populace.

11.4 Řízení generací

U generací se typicky používá náhrada, nebo náhrada s elitismem *(pozor na degeneraci)*.

Populace: malé populace = nebezpečí ztráty diverzity, **méně obtížné** problémy: **30** jedinců, **obtížné** problémy: cca **100** jedinců.

Podmínky ukončení: pevný počet generací, případně sledujeme příznaky konvergence *(pokud je lze solidně měřit)* → změna průměrné zdatnosti, rozložení zdatnosti v generaci, sledování diverzity

11.5 Omezující podmínky

Standardní: trest smrti (neúspěšná konstrukce a vyhodnocení individua), oprava individua, relaxace

Doménová reprezentace: taková, že každá možná reprezentace je platná (zobrazuje fenotyp) – např. permutační operátory = je to permutace.

Dekodér: zachová původní reprezentaci, pokud interpretujeme algoritmem, dostaneme vždy řešení *(každé řešení je reprezentováno, malá změna genotypu = malá změna řešení)*

11.6 Příklad: evoluce posouvače

Máme posouvač obvodu, který posouvá o mocniny dvou bitů, chceme zkonstruovat jednotku, která bude posouvat efektivně a bude dost malá (obvodově).

Budu hledat jen posuvy s_i pomocí genetického programování, a zbytek vypočítám pomocí dynamického programování (docela rychle).

Jednoduše se doménově reprezentuje, implementujeme pomocí jednobodového křížení → dojdeme ke konvergenci po 3 000 generacích.

11.7 Nasazení na spojité problémy

Konfiguračními proměnnými je **vektor** reálných čísel $a = (a_1, a_2, \ldots, a_n) \in \mathbb{R}^n$. Optimalizačním kritériem bude nějaká funkce z $\mathbb{R}^n \to \mathbb{R}$. Mají iterativní charakter, jak ale hledat **souseda**?

Simulované ochlazování pro spojitý problém: kromě vektoru konfiguračních proměnných máme ještě vektor s **velikostmi kroků**, operátor přičte krok v **jedné** dimenzi, pokud je posun menší, krok se zvětší, jinak se zmenší.

11.8 Evoluční strategie

Konfigurace je opět **vektor reálných čísel** a další parametry (velikosti kroků, standardní odchylky).

Kostra opět vypadá podobně, ale při selekci se zohledňují i tyto "další" parametry. Při **selekci** rozhoduje pouze pořadí **zdatnosti**.

Rekombinace: jak spojit dva vektory reálných čísel do jednoho?

Diskrétní: pro **každou** dimenzi náhodně vybrán rodič, jehož hodnota se přebírá, předpokládá se **nezávislost** hodnot v dimenzích.

Střední: všichni rodiče jsou zprůměrováni *(pokud má jeden rodič chybu, tato chyba se příliš neprojeví)*

Vážená: všichni rodiče jsou zprůměrováni váženým průměrem podle zdatnosti.

Mutace: přičtení náhodné proměnné z rozložení s **nulovou střední hodnotou**, používáme normální rozdělení *(neomezený rozsah pomůže uniknout z lokálních optim)*.

Obecně odpovídá **násobení** nějakou kovarianční maticí $\mathbb{C}$. Pro výběr mutace můžeme volit **nezávisle** na zdatnosti (deterministicky, stochasticky), pak **musí** následovat selekce na zdatnosti; nebo výběr **závislý** na zdatnosti.

11.9 Genetické programování

Hledá reprezentaci, která by umožnila **genetické operátory**, interpretaci virtuálním strojem a **vyjádření** každého (optimálního) **řešení**. Strom výrazu → genetické programování, řetězec → lineární, orientovaný acyklický graf → kartézské.

Kostra je opět víceméně stejná, ale před selekcí provádím **testy** a výběr elity *(reprodukce)*.

Stromová reprezentace: vnitřní uzly jsou operace (aritmetické, if/else), listy jsou konstanty a proměnné, musí zde platit **uzavřenost** při kompozici (netypované či typované genetické programování).

Inicializace: zvolím náhodnou operaci jako kořen a buduji náhodně předchůdce až do maximální výšky d (parametr).

Křížení: zvolím náhodný uzel v každém rodiči a prohodím podstromy.

Mutace: náhodně zvolím podstrom, ten ustřihnu a nahradím náhodně vygenerovaným.

Architektura: mám hlavní strom a různé funkce, které se vyvíjejí najednou.

11.10 Evoluční programování

Reprezentací je stavový stroj, operátory jsou změna výstupního symbolu / přechodu, přidání / vypuštění stavu, změna počátečního stavu. Řízení populace: stochastický výběr turnajem.

Není zde omezení na reprezentaci (může to být neuronová síť).

11.11 Závěrem

Operátory jsou definovány přímo na konfiguraci, nikoliv na binárním řetězci. Dokáží zachytit program jako strom, graf, nebo stavový stroj. Individuum se vyvíjí, není stálé. Nespoléhá se na kombinaci dobrých nápadů do ještě lepšího nápadu.

11.12 Proč to funguje?

Teorie **stavebních bloků**: mám dva stavební bloky (kvalitní řešení podúloh), ty vezmu a dám vedle sebe do jednoho genomu, čímž dostaneme něco ještě lepšího.

Schéma: chromozom, v němž mají některé geny neurčitou hodnotu *(odpovídá dont-cares z BI-SAP.* Řád schématu = počet určených genů, délka schematu = největší vzdálenost mezi geny v chromozomu.

Linkage learning: pozičně nezávislá notace, přeuspořádává rodiče do pořadí jiného rodiče, kříží a přeuspořádá zpět. Při hodnocení schémat pak někde nemusí být specifikované geny.

12 Globální metody

Lokální metody mají stavový prostor, aktuální stav a jeho okolí. **Globální metody** z instance problému Π vyrobí menší instance, nějak je vyřeší, a zkombinuje do výsledného řešení problému. Příkladem může být dynamické programování na problému batohu.

Při dekompozici vždy dostanu instance **menší** velikosti (typicky o 1 menší, nebo dvakrát menší). Existuje čistá dekompozice, nebo také přibližná dekompozice.

Pokud použijeme algoritmus, který používá pouze přesnou dekompozici, získáme **všechna optimální řešení**, pokud použijeme přesnou a čistou dekompozici, získáme alespoň jedno optimální řešení, jinak nemůžeme nic zaručit.

Dekompozice taková, že jedna z dekomponovaných instancí je **triviální** se nazývá **redukce**.

Specifikum některých algoritmů Rozděl a panuj může být využití přibližné dekompozice, nebo také řešení jen jedné z dekomponovaných instancí *(zmenši a panuj, binární vyhledávání).*

12.1 Dynamické programování

Dekomponované instance se dají **charakterizovat** malým objemem hodnot tak, aby to šlo **efektivně** indexovat a vyhledávat. Známe rekurzivní formulaci *(+ cachování, méně výpočtů)* a **dopřednou** formulaci *(vyplňujeme celé od triviálních řešení).*

13 Problém omezujících podmínek

Máme množinu proměnných a pro každou proměnnou **konečnou** množinu hodnot. Pak množina omezení, která jsou dvojice (množina proměnných, nějaká relace). Chceme zkonstruovat ohodnocení tak, že splňují všechny relace R_i.

Omezeními vlastně definuji **deklarativní** program. Zmenšit stavový prostor můžu postupnou **redukcí** domén (omezit na množinu unárních omezení → určitá hodnota proměnných nikdy nedovolí splnění jednoho z kritérií).

CSP se dá řešit i na **konečném intervalu**, nikoliv konečnou množinu hodnot, zůstává mnnožina omezení a opět chceme zkonstruovat ohodnocení tak, aby splnili všechny relace R_i. Tomuto se pak říká **branch and prune**.