Semestrální domácí úkol z EOA - čalouníkův problém

Tomáš Kasl

2. 1. 2022

1 Úvod

Čalouník (ale také například krejčí, švadlena, nebo dokonce truhlář) musí naplánovat střih jednotlivých kusů látky z její role. Roli si představíme tak, že délku má nekonečnou, ale šířku má striktně omezenou, většinou zhruba kolem hodnoty metru a půl (není pravidlem).
Jak tuto úlohu vyřešit optimálně?

2 Bližší specifikace problému

Ihned je jasné, že chceme minimalizovat spotřebu látky, tedy délku použité plochy role. Optimalizace tohoto kriteria pomocí evolučních algoritmů velmi snadno porazí lidského řešitele. Ten zpravidla látku rozvrhne heuristicky (optimalizuje čas), a EA nalezne výhodnější řešení.
Důležité je zmínit, že v závislosti na použité látce/kůži/kožence může záležet na tom, aby byly všechny kusy správně na látce natočené, jindy je zcela v pořádku je otočit o 90°a ušetřit více materiálu.
Záhy se ale ukáže, že toto kriterium nestačí, je potřeba optimalizovat i vedlejší kriteria, která souvisejí s pohodlností/praktičností práce nad plánem.
Konkrétně chceme minimalizovat počet jednotlivých střihů (každá manipulace s látkou na stole stojí čas). Dále chceme minimalizovat vzdálenost identických kusů na látce. Stejné kusy totiž později musí projít stejým procesem přípravy, a mít je pohromadě tak ušetří značný čas.
Máme tudíž multikriteriální optimalizaci. Vzhledem ke konkrétní implementaci ale nevalidní řešení nemají žádnou hodnotu, nemá cenu se jimi zabývat. Proto problém řeším s váženou sumou ztrátových funkcí těchto 3 kriterií memetickým algoritmem.

3 Implementace

Mé řešení je implementováno v Julii, importuji nestandardní knihovny. Chcete-li spustit prezentační skript, musíte mít na svém OS nainstalovány kromě Julie

i knihovny Plots, Distributions, Plots a Luxor.
O UI jsem se pokusil také v Julii, ale není to správný přístup. Ačkoliv Julia nabízí několik knihoven určených pro UI (zkoušel jsem práci s Interact a Blink), pro moje nároky se po hodinách zkoušení neprojevila jako použitelná žádná. Implementace je velmi zdlouhavá (částečně dána i velmi řídkou dokumentací), a runtime je pomalejší, než je v tomto roce přípustné. Úplně základn UI okno v Blink knihovně se otevírá zhruba 3 minuty.
Proto jsem se rozhodl udělat UI místo toho v Pythonu pomocí klasického Tkinteru. Pro spuštění celého programu je tedy potřeba mít kromě Julia i Python3 a Tkinter.

Implementované (varianty) algoritmy:

- Lokální prohledávání
- Základní evoluční algoritmus
- Memetický algoritmus s heuristikami
- (všechny ve verzi s/bez povoleným otočením kusů na látce)

    -¿ To proto, že s povolením otáčením vzroste stavový prostor extrémně moc, za každý z původních možných stavů je najednou $2^{početkusů}$

3.1 Reprezentace

Vstupním problémem bylo vymyslet, jak efektivně reprezentovat rozložení jakožto jedince v evolučním procesu. Prvotní nápad byl pamatovat si pro každý kus zkrátka jeho pozici v 2D prostoru (tedy mít pole dvojic $[x, y]$). Má ale 2 hlavní problémy: příliš velký stavový prostor a je těžké zjistit, kam rozumně při mutaci kus přesunout, aby něco nepřekrýval.
Další byl zhruba grid-based systém založených na sloupcích, a jedince reprezentovat jako seznam pozic v tomto systému. Bylo by ale zbytečně obtížné řešit potíže plynoucí s velké rozdílnosti velikostí jednotlivých dílů a jejich přesahů do jiných sloupců.
Nakonec jsem tedy (na doporučení) vybral reprezentaci jedince jako permutaci kusů na látce, kde kusy dle dané permutace postupně skládám zleva zdola.

3.2 Konkrétní implementace

První předmět permutace (s velikostí $[x1, y1]$ se vloží vlevo dolů, tedy na pozici $[0, 0]$. Jeho položením se vytvoří 2 nová místa pro potenciální uložení: nad něj, tj.$[0, y1]$, a vedle něj, tj. $[x1, 0]$. Ve smyčce se vezme následující kus v permutaci, a zleva doprava se vybere první místo, na které pasuje. Pak se přidají další potenciální místa na vložení. Pro optimalizaci jsou ze seznamu mazána místa, kam se s jistotou již nic nevejde, přesto tento přístup není zrovna rychlý.

3.3 Ztrátové funkce

(loss.jl)

- Délka látky: realizuji permutaci, projedu všechny kusy na látce, zjistím, který zasahuje nejvíce doprava. Tato hodnota se minimalizuje
- Počet střihů: realizuji permutaci, zjistím počet vertikálních a horizontálních střihů. Tato hodnota se minimalizuje
- Vzdálenosti identických kusů: realizuji permutaci, pro každý kus zjistím vzdálenost od dalších identických kusů, suma těchto vzdáleností se minimalizuje.

3.4 Pipeline

- Každé kandidátní řešení je dvojice <permutace, orientace>
  tj. např. $[2, 3, 1, 4], [F, T, T, F]$
- Složenou užitkovou funkci - minimalizujeme váženou sumu 3 kriterií
- Konkrétní zadání jsou vloženy uživatelem v Python UI
- Velikost populace je 150 (bez rotací)/250(s rotací), počet generací je 20
- Výběrová funkce - standardní turnaj 4 (selection_functions.jl)
- Křížicí funkce (cross_functions.jl)
  - permutace: jednobodové křížení
  - orientace: bit-or / bit-and
- Mutační funkce (mutate_function.jl)
  - Hlavní mutační funkce: Prohození 2 náhodných sekcí v permutaci
  - LC v memetickém algoritmu: Přesunutí jednoho prvku v permutaci
  - orientace: otočení kusu látky s uniformní $P = 0.25$

4 Algoritmy

4.1 Lokální prohledávání - LC

Prvním evolučním procesem je tzv. Local search, standardní lokální prohledávání. Algoritmus ve smyčce zkouší náhodné změny aplikováním mutační funkce, pokud dojde ke zlepšení ztrátové funkce, toto řešení přijme za nové pro nové mutace. Mutace s malým impaktem (například prohození 2 prvků, nebo přesunutí jednoho prvku jinam) zpravidla nedokáže uniknout z lokálního optima, proto je vybrána mutace prohození 2 celých částí permutace.

4.2 Základní evoluční algoritmus

Dalším algoritmem je standardní (naivní) evoluční algoritmus. Proces je

- Náhodně vytvoř původní populaci, tj. n jedinců (náhodné permutace)
- ve smyčce pro každou generaci:
  - Výběrovou funkcí vyber rodiče ($n/2$ párů)
  - Křížicí funkcí z párů rodičů vygeneruj potomky ($n/2$)
  - Mutační funkcí z potomků vygeneruj zmutované (dalších $n/2$)
  - Sjednoť populace (tj. $2n$ jedinců)
  - Vyber n jedinců s nejnižší hodnotou ztrátové funkce
  - Opakuj
- Je jasné, že tento algoritmus nebude pro řešení optimální
- Vede na implementaci následujícího memetického algoritmu
- Bohužel není prostor na realizaci všech nápadů

4.3 Memetický algoritmus s heuristikami

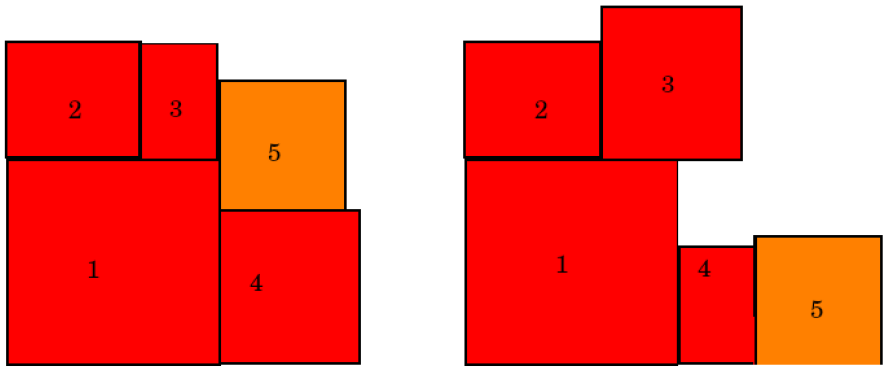
Stejný princip jako v předchozím případě, každý jedinec ale projde krátkým lokálním prohledáváním (v iniciační, křížicí i mutační fázi). Kromě toho jsou implementovány heuristiky:

- Kusy, které jsou tak vysoké, že se nad ně již nic nevejde
  (tj. $cloth\_width - y_i < min(y_k) \in 1..k$)
  jsou z procesu vyjmuty. Na jejich pozici nezáleží, protože výpočet roste polynomiálně s počtem kusů, proces tím zrychlíme.
- Původní populaci generujeme s několika jedinci reprezentujícími permutaci kusů seřazených podle výšky.

4.4 Diskuse nad výběrem funkcí evoluční pipeline

Nejdříve jsem vycházel z toho, které funkce se nejvíce osvědčily v řešení TSP v prvním domácím úkolu.
Pro výběrovou funkci jsem znovu zvolil turnaj. Má všeobecně pěkné vlastnoti:

- Výpočetně rychlý, implementačně jednoduchý
- Preferuje nejlepší jedince v populaci
- Zvolen může být i "horší" jedinec, nikdy ale ne nejhorší jedinec



3 <--> 4

Jako křížící funkci jsem prvotně vybral *edge recombination*, tedy funkci, která zachovává sousedství v permutaci, ale pořadí mění. Ukazuje se ale, že narozdíl od TSP v tomto případě nedává příliš smysl. Nejenže je výpočetně náročná, ani intuitivně se na tento problém nehodí.

V tomto případě je sice sousedství mezi bloky na 3. a 4. zachováno, výsledek je ale horší ve všech ohledech. Důležité tedy není ani tak sousedství, jako spíše pořadí jednotlivých dílů.

Podobně dopadl výběr mutační funkce. Z TSP jsem převzal mutační funkci, která otočí pořadí nějaké části permutace. Jenže výsledkem je, znovu, zachování sousedství kusů látek, ne ale pořadí. Jemnější mutační funkcí bylo v TSP prohození 2 sousedů, ale ani tato funkce nedává příliš smysl. Předpokládejme, že máme permutaci představující nějaké lokální (skoro-)optimum. Prohození 2 sousedů prakticky s jistotou vede na identické (když mutace prohodí 2 identické kusy), nebo dokonce horší řešení, k optimu se tak nepřiblížíme. Naopak zde tedy používám mutace, které mění sousednost, ale ne vzájemné pořadí.

5 Kód

Celý program se spouští Pythonovým souborem *upholsterer.py*, který zprostředuje UI pro zadání problému a spustí samostatný proces, jehož kódem je *main.jl*. Tyto procesy spolu komunikují pomocí souborů:

- *problem.txt* - přepíše zadání od uživatele do textového souboru. Tento přístup má výhody v tom, že se zadání se souborem dá zálohovat, či řešit znova spuštěním přímo *main.jl* bez využití UI
- *status.txt* - memetický algoritmus udržuje aktuální stav v souboru (pro odezvu v UI)
- *output.png* - obrázek výsledného řešení, Pythonem otevřen automaticky

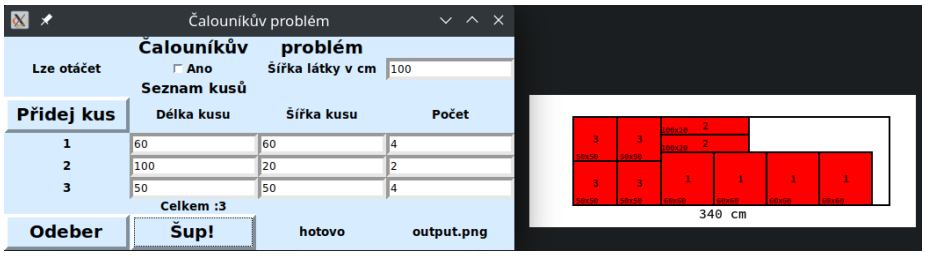


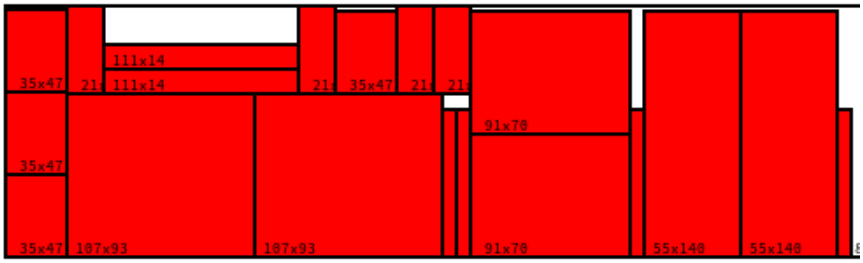
Figure 1: UI projektu

Meziprocesní komunikace pomocí souborů není nejlepší řešení, ale v tomto časovém rámci účel splní více než dostatečně.
(Téměř) celý výpočet v memetickém algoritmu se mi podařilo v Julii naimplementovat paralelně. První pokusy pomocí rozdělení výpočtů na procesy nebo subrutiny se sice nepodařily, nakonec jsem ale uspěl s paralelizací pomocí *Threads*. Paralelizace for cyklu je teoreticky velmi jednoduchá, jedním makrem Julia rozloží automaticky zátěž na více jader. Protože je to ale rozložení automatické, stálo mě několik hodin debugování zjistit všechny příčiny pádů programu. Všechna paměť musí být předem striktně prealokována, práce vláken musí být plně nezávislá.
Zrychlení výpočtu programu je na mém stroji (2C/4T) zhruba dvojnásobné, rozdíl na novějším stroji bude ještě znatelnější.

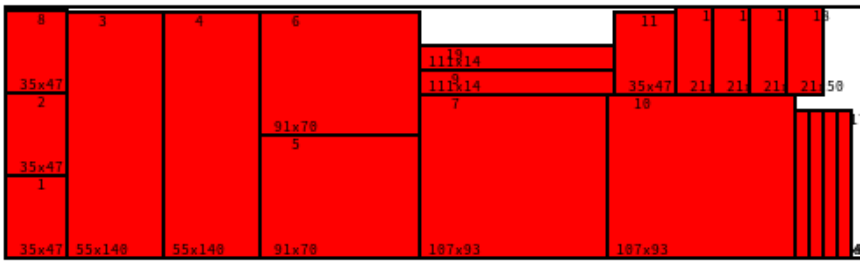
Implementované algoritmy průběžně vypisují, v jakém stavu se nacházejí, a na konci vypíší a vykreslí nejlepší nalezené řešení.

6 Ilustrace

6.1 Některé nalezené výsledky



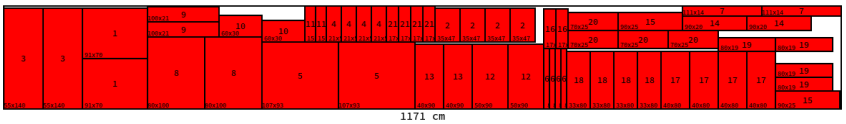
(a) Pouze délka látky



(b) Agregovaná ztrátová funkce: délka, počet střihů a shlukování

Figure 2: Porovnání nalezených výsledků podle kriteria. Celková spotřebovaná délka je stejná - 482 cm

(a) Bez otáčení - délka 1176 cm



(b) S možným otáčením - délka 1171 cm

Figure 3: Porovnání nalezených výsledků podle povoléno otáčení kusů

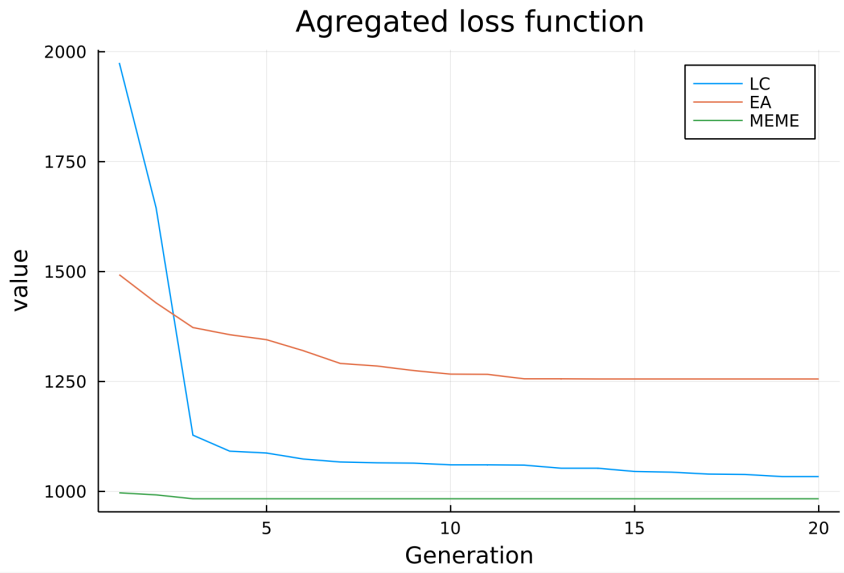


Figure 4: Porovnání konvergence algoritmů, 100 000 výpočtů ztrátové funkce, agregace za 10 běhů

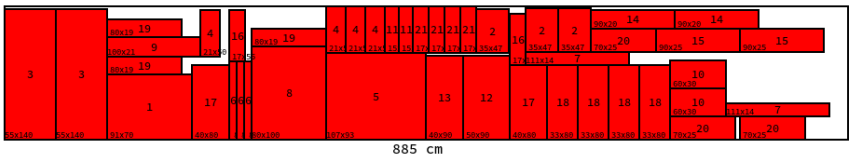


Figure 5: Zadání pro porovnání algoritmů

7 Závěr

Myslím, že se mi v rámci této práce podařilo vytvořit program, který relativně prakticky řeší tento konrétní problém a jako takový může splnit svůj účel - ušetřit čas s návrhem rozložení kusů látky na roli.
Ačkoliv jsem řešení postavil na permutacích, kódy z mého řešení prvního domácího úkolu (též postavený na permutacích) nebyly zdaleka tak použitelné, jak jsem čekal. Ukázalo se, že řešení každého problému má svá specifika (jak je dobře vidět u funkcí evoluční pipeline).
V téměr každém ohledu mojí práce je prostor k vylepšení, nicméně v omezeném čase je možné udělat jen omezené množství práce. Už jenom fakt, že tento projekt pozná opravdové nasazení v reálném světě, mě těší.

Specifické chování jednotlivých algoritmů

- Lokální prohledávání

Velmi jednoduchý na implementaci

Závislý pouze na náhodných mutacích

Konverguje pomalu

- Evoluční algoritmus

  Díky náhodné populaci začíná s lepšními jednotlivci než LC

  Kvůli relativně malému počtu počtu ale konverguje pomaleji

- Memetický algoritmus s heuristikami

  V lokálním optimu se vyskytne velmi brzo

  Většinou pomocí mutací v pozdějších generacích ještě lepší řešení nalezne

  Zdaleka nejúspěšnější řešící algoritmus

Splněno:

- Lokální prohledávání
- Evoluční algoritmus
- Specializovaný memetický algoritmus
- Základní porovnání
- Úvahy nad řešením, heuristiky
- Report
- Prezentace
- Vizualizace výsledků
- UI pro zadání úlohy
- Paralelizace

Děkuji za pozornost, Tomáš Kasl