FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Algoritmy pro řešení problému 0-1 baťohu
Student:	Jakub Pečenka
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

1) Nastudujte problém klasického 0-1 batohu [1].
2) Nastudujte přístupy k exaktnímu řešení problému klasického 0-1 batohu [1][2].
3) Implementujte zástupce z vybraných tříd přístupů řešící exaktně problém 0-1 batohu, alespoň tyto:
   - dynamické programování, algoritmus: [3]
   - metoda větví a mezí, algoritmus: [4]
   - brute force
4) Analyzujte možnosti paralelizace běhu implementací, pro vybrané implementace proveďte paralelizaci pomocí OpenMP API.
5) Porovnejte výkonnost implementací algoritmů mezi sebou a s existujícími řešeními na vybraných datasetech [5][6].

Seznam odborné literatury

[1] S. Martello, P. Toth: Knapsack problems: algorithms and computer implementations, John Wiley & Sons, Inc., New York, NY, 1990
[2] S. Martello, D. Pisinger, P. Toth: New trends in exact algorithms for the 0-1 knapsack problem (2000) European Journal of Operational Research, 123 (2) , pp. 325-332.
[3] S. Martello, P. Toth, An upper bound for the zero-one knapsack problem and a branch and bound algorithm, European Journal of Operational Research 1 (1977) 169-175.
[4] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, Journal of ACM 21 (1974) 277-292.
[5] http://artemisa.unicauca.edu.co/ johnyortega/instances_01_KP/
[6] http://hjemmesider.diku.dk/ pisinger/codes.html

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 9. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Algoritmy pro řešení problému 0-1 batohu

Jakub Pečenka

Katedra teoretické informatiky
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

14. května 2019

Poděkování

Děkuji panu doc. Ing. Ivanu Šimečkovi, Ph.D. za rady a čas, který mi věnoval.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. května 2019 .....

Odkaz na tuto práci

Pečenka, Jakub. *Algoritmy pro řešení problému 0-1 batohu.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato Bakalářská práce se zabývá problémem batohu 0-1 a přístupy hledajícími jeho optimální řešení. V práci je implementován v jazyce *C++* algoritmus založený na metodě *větví a mezí*, dále algoritmus založený na metodě *dynamického programování* a metoda *hrubé síly*. Autor se v práci následně zabývá jejich paralelizací a vybrané paralelizace implementuje pomocí technologie *OpenMP*. Na závěr jsou změřeny a porovnány výpočetní časy těchto implementací a implementací konkurenčních na vybraných datových sadách.

Klíčová slova batoh 0-1, dynamické programování, větve a meze, hrubá síla, paralelizace, implementace, C++, OpenMP

Abstract

This thesis focuses on problem Knapsack 0-1 and approaches for searching exact solution. Choosed algorithms based on method *branch and bounds*, on method *dynamic programming* and *brute force* method are implemented in *C++* language. After that author focuses on paralelization of that algorithms and implementation of paralelizations by *OpenMP*. Finally for authors implementations and for competitive implementations are measured and compared computational times on choosed data sets.

Keywords knapsack 0-1, dynamic programming, branch and bounds, brute force, paralelization, implementation, C++, OpenMP

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

Úvod

Problém batohu 0-1 patří mezi nejznámější diskrétní optimalizační problémy. Má mnoho praktických aplikací a v současné době rozvoje více jádrových systémů má smysl zabývat se paralelizací metod řešících tento problém.

V úvodu je definován samotný problém batohu 0-1. Dále text nastiňuje techniky používané k hledání exaktního řešení problému batohu 0-1.

Poté jsou popsány a implementovány dva vybrané sekvenční algoritmy, které jsou součástí pokročilejších algoritmů. Jeden stojící na metodě *větví a mezí* a druhý stojící na metodě *dynamického programování*, pro který je také implementováno rozdělení instance batohu 0-1 na dvě nezávislé části s následným nalezením optimálního řešení. Autor se také zabývá možností rozdělení instance batohu 0-1 na více nezávislých částí. Dále je implementována metoda *hrubé síly*. Pro tyto algoritmy je navrhnuta a implementována paralelizace.

Na závěr jsou změřeny výpočetní časy implementací autora a také implementací konkurenčních na vybraných datových sadách. Následně jsou diskutovány dosažené výsledky.

Kapitola 1

Cíl práce

Cílem teoretické části práce je popis problému batohu 0-1, představující kombinatorický optimalizační problém. Dále popsání přístupů používaných k nalezení exaktních, respektive optimálních řešení. Cílem je také popsat vybrané algoritmy a přístup metodou hrubé síly.

Cílem praktické části práce je tyto vybrané algoritmy implementovat, navrhnout jejich paralelizaci a implementovat paralelní verze pomocí technologie *OpenMP*. Dále je cílem praktické části práce porovnaní výpočetních časů těchto implementací mezi sebou a s konkurenčními implementacemi na vybraných datových sadách.

Kapitola 2

Batoh 0-1

V této kapitole budou nejprve zavedeny obecné pojmy používané dále v textu. Dále popsán problém batohu 0-1 a přístupy k jeho exaktnímu řešení.

Instance problému je konkrétní případ nějakého problému. Například konkrétně definovaný případ problému batohu 0-1.

Optimální řešení / Optimum představuje nejlepší možné řešení instance problému.

Kombinatorická optimalizace se zabývá hledáním optima v konečné množině možných řešení.

Účelová funkce ohodnocuje řešení daného problému podle stanovených kritérií. Tedy řešení je optimální, pokud mu účelová funkce přiřadí ohodnocení, které je maximální, pokud je řešen problém maximalizace, respektive minimální, pokud je řešen problém minimalizace.

Omezující podmínky jsou omezení kladená na výsledná řešení. Omezení mohou být na úrovni instance problému nebo na úrovni problému samotného.

Prostor přípustných řešení je množina řešení splňující omezující podmínky. Taková řešení se označují jako přípustná.

Prostor nepřípustných řešení je množina všech řešení mimo těch obsažených v prostoru přípustných řešení. Taková řešení se označují jako nepřípustná.

Optimalizace s omezujícími podmínkami představuje hledání optima účelové funkce v prostoru přípustných řešení.

Optimalizační model dle [1] definuje hlavní části optimalizačního problému, které jsou:

1. **Optimalizační proměnné**, u nichž jsou hledány hodnoty, pro které je účelová funkce optimální.
2. **Definiční obor proměnných**, který určuje, jakých hodnot proměnné mohou nabývat.

3. **Účelová funkce**, která závisí na proměnných a pro kterou je hledáno optimální řešení.

4. **Omezení**, která jsou reprezentovány funkcemi, které závisí na proměnných, a konstantami, které vyjadřují míru daného omezení. Dohromady vyjadřují omezující podmínky vztahující se na nalezená řešení.

Stavový prostor, je množina stavů definována vstupním stavem, koncovými stavy, mezistavy a přechody mezi nimi. Procházení stavového prostoru za účelem nalezení určitého stavu se nazývá *prohledávání stavového prostoru.*

Rozhodovací problém je takový problém, jehož řešení je buď *ano* nebo *ne*.

Třída NP je třída rozhodovacích problémů které je možné vyřešit v na nedeterministickém Turingově stroji v polynomiálně omezeném čase.

Pseudo polynomální složitost je výpočetní čas polynomiální vzhledem k číselné velikosti vstupu, ale ne k počtu bitů nutných k zakódování vstupu.

Míra korelace vyjadřuje lineární vztah mezi dvěma veličinami. Míra korelace je vyjádřena korelačním koeficientem, který nabývá hodnot v intervalu $< 1, -1 >$. Hodnoty blíže nule znamenají menší korelaci a naopak.

2.1 Popis problému batohu 0-1

Neformální znění problému batohu 0-1 lze uvést následovně:

Nechť máme batoh o kapacitě c a n věcí, každou o určité hodnotě a velikosti. Poté hledáme, které věci do batohu umístit tak, aby součet jejich hodnot byl co největší a zároveň součet jejich velikosti nepřesáhl kapacitu batohu.

Z neformální definice problému batohu implicitně vyplývá, že hodnoty velikosti a zisku prvků jsou nezáporné, stejně tak kapacita batohu. Problém batohu explicitně takováto omezení nestanovuje a hodnoty mohou být záporné. Nicméně tradiční praktické aplikace batohu 0-1 a literatura ohledně batohu 0-1 předpokládají jen prvky s kladnými hodnotami zisku, váhy a kapacity.

Problém batohu má uplatnění v mnoha problémech reálného světa zabývajících se maximalizací užitku z omezených zdrojů. Například:

1. Investice s omezeným kapitálem na burze, kde investice mají cenu a očekávaný zisk.

2. Vyklizení skladu zásob, které mají hodnotu a velikost, před přírodní katastrofou, pokud máme omezenou kapacitu k přepravě těchto zásob.

2.1.1 Problém batohu 0-1 jako kombinatorický optimalizační problém

V této sekci bude vycházeno z [2].

Problém je možné definovat ve dvou verzích, a to maximalizační a minimalizační, které jsou na sebe vzájemně převeditelné. Problém se skládá z následujících komponent:

- $w = (w_1, \ldots, w_n)$ - vektor vah,
- $p = (p_1, \ldots, p_n)$ - vektor zisků,
- $x = (x_1, \ldots, x_n), x \in \{0, 1\}$ - vektor přítomnosti,
- f_0 - účelová funkce,
- f_1 - omezující funkce,
- c - konstanta představující mez omezující funkce f_1.

Poté i-tá souřadnice vektoru vah určuje množství kapacity batohu zabrané i-tým prvkem, dále i-tá souřadnice vektoru zisků určuje hodnotu i-tého prvku a i-tá souřadnice vektoru přítomnosti určuje přítomnost i-tého prvku v batohu. Vektory zisků a vah jsou pevně dané parametry pro danou instanci problému.

Maximalizační verze batohu 0-1 je definována následovně:
Hledáme vektory přítomnosti x takové, že

$$f_1(x, w) = \sum_{i=1}^{n} x_i w_i \leq c$$

a

$$f_0(x, p) = \sum_{i=1}^{n} x_i p_i$$

je maximální.

Minimalizační verze batohu 0-1 je definována následovně:
Hledáme vektory přítomnosti x takové, že

$$f_1(x, w) = \sum_{i=1}^{n} x_i w_i \geq q,$$

kde q je minimální váha prvků umístěných do batohu a

$$f_0(x, p) = \sum_{i=1}^{n} x_i p_i$$

je minimální.

Převod mezi maximalizační a minimalizační verzí je dán následovně:
Nechť jsou dány:

- vektor přítomnosti v maximalizačním problému
  $x = (x_1, \ldots, x_n), x \in 0, 1,$
- vektor přítomnosti v minimalizačním problému
  $y = (y_1, \ldots, y_n), y \in 0, 1,$
- optimální řešení maximalizačního problému z_{max},
- optimální řešení minimalizačního problému z_{min}.

Minimalizační na maximalizační problém je možné převést pomocí rovnice $c = \sum_{i=1}^{n} w_i - q$, kdy je následně řešen maximalizační problém. Optimální řešení je dáno vztahem $z_{min} = \sum_{i=1}^{n} p_i - z_{max}$ a vektor přítomnosti je získán jako $y_j = 1 - x_j$.

Maximalizační na minimalizační problém je možné převést pomocí rovnice $q = \sum_{i=1}^{n} w_i - c$, kdy je následně řešen minimalizační problém. Optimální řešení je dáno vztahem $z_{max} = \sum_{i=1}^{n} p_i - z_{min}$ a vektor přítomnosti je získán jako $x_j = 1 - y_j$.

Dle [2] se lze zabývat bez újmy na obecnosti pouze maximalizační verzí s kapacitou a atributy prvků s celočíselnými kladnými hodnotami. Neboť maximalizační a minimalizační forma jsou na sebe převoditelné. Neceločíselné hodnoty lze ošetřit vynásobením vhodným činitelem a při záporných hodnotách lze prvky rozdělit do následující tříd:

1) $w_j < 0$ *a* $p_j < 0$, 2) $w_j > 0$ *a* $p_j > 0$, 3) $w_j = 0$ *a* $p_j = 0$,
4) $w_j < 0$ *a* $p_j > 0$, 5) $w_j < 0$ *a* $p_j = 0$, 6) $w_j = 0$ *a* $p_j > 0$,
7) $w_j > 0$ *a* $p_j = 0$, 8) $w_j = 0$ *a* $p_j < 0$, 9) $w_j > 0$ *a* $p_j < 0$,

kde čísla 1-9 jsou jednotlivé třídy. Pro prvky z třídy 1, 2 platí, že zařazení těchto prvků do optimálního řešení vyplyne z řešení instance problému, neboť nelze jednoznačně rozhodnout o jejich přínosu bez znalosti kontextu v podobě dalších prvků. Pro prvky z třídy 3 platí, že nemají vliv na řešení instance problému, a nemusí tedy být při řešení uvažovány. Pro prvky z třídy 4, 5, 6 platí, že budou do řešení zahrnuty vždy, neboť jejich přínos je čistě kladný. Buď jen přidávají zisk, nebo jen snižují zaplněnost batohu nebo obojí. Pro prvky z třídy 7, 8, 9 platí, že nebudou do řešení nikdy zahrnuty, neboť jejich přínos je čistě záporný. Buď jen snižují zisk, nebo jen zabírají kapacitu batohu nebo obojí.

Složitost: Dle [2] je hledání optimálního řešení pro problém batohu 0-1 NP těžký problém a existují algoritmy využívající dynamického programování s pseudo-polynomiální složitostí.

2.1.2 Typy instancí problému

Dle [3] obtížnost řešení instance problému batohu 0-1 ovlivňuje míra korelace mezi atributy prvků, tedy mezi vahou a ziskem a rozsah hodnot kterých mohou nabývat. Za méně obtížné jsou považovány případy s menší mírou korelace mezi atributy prvků a s menšími rozsahy hodnot. S vzrůstající mírou korelace a rozsahem hodnot vrůstá i obtížnost dané instance problému, neboť klesá efektivnost technik redukujících stavový prostor řešení.

2.2 Přístupy k exaktnímu řešení batohu 0-1

Počet všech možných řešení je 2^n, kde n je počet prvků, stavový prostor tedy roste exponenciálně k velikosti vstupu. S rostoucí velikostí vstupu je časově náročné procházet všechna řešení. Přistupuje se tedy buď k odhadům optimálního řešení pomocí aproximativních algoritmů, nebo metodám, které dokáží najít optimální řešení v přijatelném čase. Dále se před nebo při samotném hledáním optimálního řešení mohou aplikovat metody, které se o přítomnosti prvků v optimálním řešení snaží rozhodnout na základě výpočetně nenáročných postupů.

V exaktním přístupu jsou hledány taková řešení, která splňují omezující podmínky a jsou optimální, tedy mají maximální účelovou funkci. Dle [4] jsou k tomuto využívány převážně následující metody, které budou kromě následujícího výčtu ještě v textu dále přiblíženy. Používané metody jsou:

- redukční metody,
- metoda větví a mezí,
- dynamické programování a dynamické programování s horní mezí,
- problém jádra.

Dle [4] lze jako úspěšný algoritmus stojící na metodě větví a mezí uvést algoritmus *MTHard* popsaný v [5] a jako úspěšný algoritmus stojící na dynamickém programování s horní mezí lze uvést algoritmus *minknap* popsaný v [6]. Dále dle [3] je neúspěšnější algoritmus *combo* uvedený v [7], tento algoritmus využívá také dynamické programování s horní mezí. Všechny tyto metody využívají problém jádra.

Pro porovnání výpočetních časů výše zmíněných a dalších algoritmů autor čtenáře v případě zájmu odkazuje do [4], [8] a [3].

2.2.1 Redukční metody

Autor v této části vychází z [2].

Ke zmenšení velikosti instance problému se používají redukční metody, které se snaží určit hodnoty optimalizačních proměnných v optimálním řešení

před samotným řešením instance problému. První redukční algoritmus je popsán v [9], který pro prvek, jehož přítomnost v optimálním řešení je určována, hledá horní mez variant, kdy je nebo není obsažen v řešení. Pokud je horní mez jedné varianty horší než nějaké řešení celé instance problému, je hodnota optimalizační proměnné reprezentující daný prvek nastavena na hodnotu druhé varianty. Stejně je postupováno, pokud jedna z variant vede na nepřípustně řešení. Následně se je třeba zabývat jen prvky, pro které nebyla určena přítomnost v optimálním řešení redukčním algoritmem.

2.2.2 Metoda větví a mezí

V popisu metody větví a mezí autor vycházel z [10] a [11], pokud nebude uvedeno jinak.

Metoda větví a mezí spočívá v prohledávání stavového prostoru, respektive množiny všech řešení pomocí dělení (větvení) této množiny na podmnožiny na základě přidávání omezujících podmínek. Tyto nové omezující podmínky jsou většinou reprezentovány stanovením hodnot optimalizačních proměnných. Algoritmus končí po prohledání všech řešení, což může nastat přímo explicitním prohledáním nebo vyloučením z prohledávání. Pokud množina všech přípustných řešení není nekonečná a generované podmnožiny mají vždy menší velikost než množina ze které byly vyděleny, poté tato metoda najde optimální řešení v konečném čase.

Pomocí teorie grafů se tato metoda reprezentuje jako zakořeněný strom. Základní pojmy pojící se ke grafové reprezentaci dle [1] jsou:

- **Vrchol:** Reprezentuje nějakou množinu řešení. Vrcholy lze rozdělit na 3 druhy:
  - **Kořen:** Reprezentuje množinu všech řešení, tedy bez omezujících podmínek.
  - **List:** Reprezentuje jedno konkrétní řešení jednoznačně určeno omezujícími podmínkami na cestě od tohoto listu ke kořeni. Tato cesta se nazývá větev stromu.
  - **Mezilehlý vrchol:** Není kořenem ani listem. Je to podmnožina množiny všech řešení s velikostí větší než jedna, která je určena omezujícími podmínkami vztahujícími se k tomuto vrcholu.
- **Větev:** Představuje cestu od kořene k listu a reprezentuje jedno konkrétní řešení.
- **Nerozvětvený vrchol:** Jedná se o vrchol, který může být dále rozvětven. Je to buď kořen nebo mezilehlý vrchol.
- **Funkce stanovující mez:** Pro vrchol stanovuje optimistický odhad hodnoty účelové funkce, které je možné dosáhnout.

- **Větvení vrcholu:** Stanovuje způsob větvení vrcholu. Má dvě hlavní části a to, jak vybrat proměnnou použitou k větvení a samotný způsob větvení.

- **Dosud nejlepší nalezené řešení:** Pokud je nalezeno řešení, je porovnáno s dosavadním nejlepším nalezeným řešením, pokud je lepší je dosavadní nejlepší řešení tímto řešením nahrazeno. Pokud není počáteční hodnota explicitně vypočítána je iniciální hodnota $+\infty$ u minimalizace a $-\infty$ u maximalizace.

Nyní budou popsány části algoritmu, autor bude vycházet z [10] a [11]:

- **Větvení:** Větvení určuje postup, jak dělit množinu řešení na dvě a více podmnožin, kdy podmnožina musí splňovat omezení svých předků spolu s omezující podmínkou na základě které byla vydělena z její nadmnožiny. S každým dalším dělením se díky přibývajícím omezením množiny řešení zmenšují a stávají se více specifickými. Konkrétní strategie závisí na problému, na který je algoritmus uplatněn. Maximální počet potomků vytvořených z jednoho vrcholu se nazývá *větvící faktor*.

- **Výběr proměnné k větvení:** K rozdělení množiny řešení je třeba vybrat proměnnou, podle které dělení množiny na podmnožiny bude probíhat.

- **Prohledávání:** Pokud existuje více vygenerovaných podmnožin, ve kterých se může nacházet optimální řešení, je třeba rozhodnout, kterou podmnožinu vybrat k dalšímu prohledávání. Základní techniky používané pro prohledávání stavového stromu při technice větví a mezí jsou:

  - best first search,
  - depth firt search,
  - breadth first search,
  - cyclic best first search.

- **Stanovování mezí:** Pro hledání meze slouží funkce stanovující mez, která odhadne, jaké nejlepší hodnoty účelové funkce daná množina řešení může nabývat. Mez je určována jen u vrcholů, které nemají potomky, neboť slouží k jejich případnému vyloučení z dalšího prohledávání. Aby nebyla vyloučena množina řešení s optimálním řešením je důležité, aby funkce stanovující mez byla optimistická, tedy aby množinu řešení ohodnotila vždy lépe než účelová funkce, ale zároveň by měla být co nejpřesnější, což je často vykoupeno rostoucím výpočetním časem. Pro listy by se hodnota účelové funkce měla rovnat hodnotě funkce určující meze, neboť je dáno právě jedno jednoznačné řešení. Pro nadmnožinu by ohodnocení mělo být lepší nebo stejné jako ohodnocení její podmnožiny.

- **Prořezávání:** Pokud je dán například větvící faktor dva, počet vrcholů prohledávacího stromu při n prvcích bude $\sum_{i=1}^{n} 2^i$ vrcholů. Z důvodu takto velkého počtu vrcholů se přistupuje k technice prořezávání, která redukuje velikost stavového prostoru nutného k prohledání.

  Prořezávání může nastat na základě 3 skutečností:

  - **Prořezávání na základě splnitelnosti množiny řešení:** Pokud všechny řešení z dané množiny porušují omezující podmínky kladené na řešení, potom může být tato množina vyloučena z dalšího prohledávání.
  - **Prořezávání na základě dosavadního nejlepšího nalezeného řešení a meze dané množiny řešení:** Pokud je stanovené dosavadní nejlepší řešení je možné vyloučit z dalšího prohledávání množiny řešení jejichž horní mez je horší než toto řešení. Prořezávání je možné, neboť horní mez dané množiny říká optimistický odhad nejlepšího možného výsledku účelové funkce dané množiny. Pokud je tato horní mez horší než dosavadní nejlepší nalezené řešení, potom je patrné, že v dané množině se nenachází lepší řešení a není třeba se touto množinou dále zabývat. Často je nalezeno počáteční nejlepší řešení pomocí heuristiky, neboť umožňuje toto prořezávání již od začátku běhu algoritmu.
  - **Prořezávání na základě dominance jedné množiny řešení nad druhou:** Tento typ prořezávání nastává, pokud jedna množina řešení dominuje jinou, tedy pokud pro každé řešení z jedné množiny existuje lepší nebo stejné řešení z množiny druhé.

Dle [10] je, bez uplatnění technik prořezávání, které značně zlepšují časovou náročnost algoritmu, asymptotická složitost $\mathcal{O}(M \cdot b^d)$, kde b je větvící faktor, d je maximální délka větve a M je maximální čas nutný ke zpracování jednoho vrcholu.

2.2.2.1 Metoda větví a mezí v kontextu batohu 0-1

Pro batoh 0-1 optimalizační proměnné nabývají pouze dvou hodnot, a to buď 0 nebo 1, větvící faktor je tedy 2. Buď je přítomnost prvku v batohu vyloučena nebo přikázána. Většina algoritmů tohoto typu pracuje s *efektivitou prvků*, respektive poměrem jejich velikosti a zisku. Tento poměr je možné využit při výběru proměnných k větvení, stanovení mezí apod.

Dle [12] je v každé množině řešení možné jednotlivé prvky klasifikovat podle omezujících podmínek jako:

- zahrnuté v řešení,
- vyloučené z řešení,

- jako prvky o kterých dosud nebylo nerozhodnuto a nejsou tedy ani zahrnuté ani vyloučené.

Dále pokud množina obsahuje jen prvky, které jsou určené omezujícími podmínkami, obsahuje tato množina právě jedno řešení.

Dle [8] je tradiční horní mezí tzv. *continous knapsack problem* spočívající v relaxaci podmínky na celočíselnost vektoru přítomnosti. Při seřazení prvků sestupně dle efektivity je optimální řešení dané umístěním do batohu co nejdelšího souvislého intervalu prvků od nejefektivnějšího, zbývající kapacita je zaplněna efektivitou prvního prvku, který se již celý nevešel do batohu. Tento první prvek, který se již do batohu nevešel, se nazývá *kritický prvek*.

2.2.3 Dynamické programování

Před popsáním metody dynamického programování je třeba popsat metodu ze které vychází, a to metodu rozděl a panuj.

Rozděl a panuj

Základním stavebním kamenem metody *rozděl a panuj* je dle [13] nalezení rekurentního vztahu, kdy z řešení podproblémů je skládáno řešení celého problému. Metoda se dle [13] skládá z následujících kroků:

1. **Rozděl:** Rozdělí problém na menší podproblémy. Je výhodné, pokud podproblémy mají podobnou, nejlépe stejnou velikost.

2. **Panuj:** Vyřeší vytvořené podproblémy z předchozího kroku. Buď rekurzivně vytvořením dalších podproblémů, nebo přímo pokud algoritmus došel k triviálnímu podproblému, ke kterému je známé řešení bez ohledu na instanci problému, nebo jiným algoritmem, pokud je vhodnější než další rekurzivní dělení.

3. **Slož:** Složí řešení problému z podproblému vydělených z tohoto problému a případně z části tohoto problému nevyděleného do podproblémů vyřešenou na úrovni tohoto problému.

Dynamické programování

Při popisu dynamického programování bude vycházeno z [14].

Pokud se při dělení problému na podproblémy objevují ve významném množství opakující se podproblémy, pak je metoda *rozděl a panuj* řeší pokaždé znovu. Neboť toto opakované řešení stejných podproblémů může významně ovlivnit efektivitu algoritmu, přistupuje se v takovém případě k modifikaci této metody o zapamatování si již vyřešených podproblémů. Díky tomu je každý podproblém řešen maximálně jednou a následně je jeho řešení uloženo pro další použití. Tato modifikace se nazývá *dynamické programování*. Cenou za tuto

vyšší efektivitu je větší paměťová náročnost a režie spojená s uchováváním již vyřešených podproblémů.

Pro využití metody dynamického programování je třeba, aby řešený problém měl vlastnost *optimal substructure*, která říká že optimální řešení problému je obsaženo v optimálních řešeních jeho podproblémů. Tato podmínka platí i pro metodu *rozděl a panuj*. K vlastnosti *optimal substructure* je u dynamického programování přidána podmínka na opakování stejných podproblémů neboli podmínka, že problém obsahuje takzvané *překrývající se podproblémy*.

Je výhodné, pokud je problém rozložen na menší množství často se opakujících podproblémů. Poté dynamické programování využije svoji hlavní výhodu v podobě ukládání si již vyřešených podproblémů. Pokud by naopak opakujících se podproblémů bylo zanedbatelné množství, potom by režie spojená s dynamickým programováním mohla převážit přidanou hodnotu oproti metodě rozděl a panuj.

Dva základní přístupy se stejnou asymptotickou složitostí jsou:

- Metoda *zdola nahoru*, kdy se postupuje od nejmenších podproblémů k větším, ze kterých se skládá postupně celé řešení. Nevýhoda tohoto přístupu spočívá v řešení všech podproblémů, tedy i těch které nejsou k nalezení řešení problému potřeba.

- Metoda *shora dolů*, kdy se začíná od celého problému, který se dělí na menší podproblémy, z jejichž řešení skládá řešení sebe sama. Tedy jsou řešeny jen podproblémy nutné k nalezení řešení problému. Nevýhodou tohoto přístupu je větší režie s uchováváním hodnot již vyřešených podproblémů a větší režie spojená s řešením jen určitých podproblémů.

2.2.3.1 Dynamické programování v kontextu batohu 0-1

V této sekci bude vycházeno z [15].

Batoh 0-1 má vlastnost *optimal substructure*, neboť odstraněním libovolného prvku z optimálního řešení zbude optimální řešení pro zredukovanou instanci batohu o kapacitě c - w_n a s prvky zvažovanými k umístění do řešení jako originální problém, ale bez prvku x_n. Pokud by tomu tak nebylo, optimální řešení by nebylo optimální, neboť zlepšením řešení redukované instance by bylo zlepšeno i původní optimální řešení, které ale tedy nemohlo být optimální. Na tomto faktu je založen přístup k řešení batohu 0-1 pomocí dynamického programování, kdy k podmnožině prvků $\{x_1, ..., x_{j-1}\}$, pro kterou je již známé optimální řešení, je přidán další prvek x_j a pro tuto novou podmnožinu prvků $\{x_1, ..., x_{j-1}, x_j\}$ je nalezeno optimální řešení. Vztah mezi optimálními řešeními těchto dvou podmnožin je možné zapsat pomocí *Bellmanovy rekurze* uvedené v [16] následovně

$$z_j(d) = \begin{cases} z_{j-1}(d) & pokud\ d < w_j, \\ max\{z_{j-1}(d), z_{j-1}(d - w_j) + p_j\} & pokud\ d \geq w_j, \end{cases}$$

kde $z_j(d)$ představuje optimální řešení pro množinu prvků $\{x_1, ..., x_{j-1}, x_j\}$ a kapacitu batohu d. Pokud $d < w_j$, potom přidávaný prvek má větší velikost než je zvažovaná kapacita batohu, nemusíme ho tedy uvažovat. Pokud $d \geq w_j$, potom je třeba rozhodnout jestli prvek do batohu umístit nebo ne, aby řešení bylo optimální.

Podle výše uvedeného je možné zkonstruovat tradičně uváděný algoritmus *Dynamic programming by weights* využívající metody dynamického programování zdola nahoru. Iniciálně $z_0(d) = 0$ *pro* $d = 0, ..., c$, kde c je kapacita batohu. Dále algoritmus iteruje od $j = 1$ do $j = n$, kde n je počet prvků. V iteraci jsou zvažovány prvky $1...j$ pro které je hledáno optimální řešení pro $d = 0, ..., c$. Je tedy konstruována pomyslná tabulka s n sloupci a c řádky.

Tento algoritmus má asymptotickou časovou a paměťovou složitost $\mathcal{O}(n \cdot c)$, kde n je počet prvků a c je kapacita batohu. Řeší tedy problém batohu 0-1 v pseudo-polynomiálním čase. Bez požadavku na konfiguraci výsledného řešení lze algoritmus modifikovat tak, že má paměťovou složitost $\mathcal{O}(c)$.

Kombinace dynamického programování s horní mezí bude popsána v kapitole 3.2.4.

2.2.4 Problém jádra

Dle [15] tato metoda stojí na myšlence, že prvky s vysokou efektivitou mají vyšší pravděpodobnost umístění do batohu než prvky s efektivitou nízkou, respektive že pravděpodobnost přítomnosti prvků s vysokou efektivitou v optimálním řešení je skoro jistá a prvků s nízkou efektivitou takřka vyloučena. O přítomnosti prvku neřadících se ani do jedné z kategorií je třeba rozhodnout. Výhoda řešení problému jádra spočívá ve skutečnosti, že je většinou nalezeno dobře využitelného řešení k prořezávání, a zafixování přítomnosti prvků v řešení pomocí redukčních algoritmů.

Autor pro popis jádra a problému jádra vycházel z [17], kde byly dané pojmy definovány. Nechť jsou prvky seřazené podle efektivity a je dán vektor přítomnosti vyjadřující optimální řešení. Dále nechť

- j_0 = první 0 ve vektoru přitomnosti,
- j_1 = poslední 1 ve vektoru přitomnosti.

Potom interval $[j_0, j_1]$ pokud $j_0 \leq j_1$, nebo $[j_1, j_0]$ pokud $j_1 \leq j_0$ je jádrem instance problému batohu 0-1.

Problém jádra je definován jako řešení instance problému, kde prvky před j_0 jsou zahrnuty v řešení, prvky za j_1 nejsou zahrnuty v řešení a o umístění prvků z intervalu $[j_0, j_1]$ do řešení musí být rozhodnuto.

Vzhledem k tomu, že optimální vektor přítomnosti není znám před vyřešením dané instance problému se dle [15] přistupuje se k odhadu jádra v čase $\mathcal{O}(n)$.

2.2.5 Metoda hrubé síly

Metoda hrubé síly spočívá ve prohledání všech řešení, což zaručuje nalezení správného výsledku. Časová složitost je vázána na počet všech řešení, který může stoupat velice rychle s rostoucí velikostí problému. S prostým prohledáním všech řešení se pojí lehká implementace. Tento přístup lze využít jako dolní mez pro hodnocení výkonnosti jiných algoritmů pod kterou by neměli klesnout.

2.2.5.1 Metoda hrubé síly v kontextu batohu 0-1

Pro implementaci metody hrubé síly pro problém batohu 0-1 se lze inspirovat algoritmem generujícím všechny kombinace z písmem nějakého řetězce. Tedy převedeno na problém batohu 0-1, všechny kombinace prvků. Dále není nutné přidávat další prvky do řešení překračující kapacitu batohu, čímž dojde k určité redukci stavového prostoru.

Kapitola 3

Popis vybraných algoritmů

Pokročilé algoritmy pro řešení problému batohu 0-1 využívají kombinaci různých metod, nicméně je jejich součástí metoda větví a mezí nebo dynamické programování.

Jako zástupce metody větví a mezí si autor pro implementaci a paralelizaci vybral algoritmus *MT1*, který je využit v algoritmu *MT2* popsaném v [18], který je využit algoritmu *MTHard* zmíněném v kapitole 2.2.

Dále si autor vybral algoritmus stojící na metodě dynamického programování popsaný v [19], na který bude autor v textu odkazovat pod zkratkou *HS1974*, který přináší rozdělení jedné instance batohu na více nezávislých instancí, které jsou řešeny odděleně a je proto zajímavý z hlediska paralelizace. Samotné dynamické programování je možné implementovat metodou *Dynamic programming with lists* popsanou v [15], autor se na tento algoritmus bude v textu odkazovat zkratkou *DPwL*. Algoritmus *DPwL* je využit jako součást algoritmů *minknap* a *combo* zmíněných v kapitole 2.2.

3.1 Algoritmus *MT1*

Následující algoritmus je popsán v [20] a autor z tohoto zdroje vycházel. Tento algoritmus stojí na metodě větví a mezí, využívá prohledávací strategií *depth first search*, dále na ní bude odkazováno pod zkratkou *DFS*, a horní meze popsané níže.

První krok algoritmu spočívá v seřazení prvků sestupně podle efektivity, v textu bude toto seřazení uvažováno jako sestupné zleva doprava, kdy první prvek zleva bude uvažován jako první a první prvek zprava jako poslední. Tento krok přináší časovou složitost $\mathcal{O}(n \cdot log(n))$, kde n je velikost vstupu. Poté je pro každý prvek zaznamenána nejmenší hodnota váhy prvků od něj napravo.

Algoritmus přichází s novou horní mezí, která se od horní meze získané řešením *continous knapsack problem* liší úvahou, že optimálního řešení může být dosaženo umístěním nebo neumístěním kritického prvku do batohu, ne-

chť je kritický prvek označen x_{l+1}. Tedy pokud prvek x_{l+1} nebude umístěn do řešení, je nejlepší potencionální řešení dáno vyplněním zbývající kapacity efektivitou prvku x_{l+2}:

$$B_1 = \sum_{j=1}^{l} p_j + [(c - \sum_{j=1}^{l} w_j) \cdot p_{l+2}/w_{l+2}]$$

Pokud prvek x_{l+1} bude umístěn do řešení, je třeba mu uvolnit místo odstraněním části předcházejícího prvku:

$$B_2 = \sum_{j=1}^{l} p_j + [p_{l+1} - (w_{l+1} - (c - \sum_{j=1}^{l} w_j)) \cdot p_l/w_l]$$

Větší z $B1$ a $B2$ je horní mez. Korektnost horní meze plyne ze skutečnosti, že do řešení byly umisťovány prvky s největší efektivitou a tedy nemůže být nalezeno lepší řešení. Tato mez je vždy menší nebo rovna mezi získané pomocí *continous knapsack problem*, tedy tento přístup určuje horní mez přesněji. Pro důkaz autor čtenáře odkazuje do uvedeného zdroje.

V průběhu algoritmu jsou prováděny následující testy:

- Test na horní mez celé instance problému / test 1: Pokud účelová funkce ohodnotí řešení hodnotou rovné horní mezi instance problému, bylo nalezeno optimum a není třeba pokračovat v prohledávání.

- Test na horní mez 1 / test 2: Je dáno budované řešení A a prvek x, který může být umístěn do řešení A. Je otestováno, jestli řešení B vzniklé z řešení A vyplněním zbývající kapacity efektivitou prvku x zlepší dosavadní nejlepší řešení. Protože každý prvek za prvkem x má stejnou nebo horší efektivitu, je řešení B nejlepší řešení jakého lze potencionálně dosáhnout s právě budovaným řešením. Tato horní mez je porovnána s dosavadním nejlepším řešením. Pokud je dosavadní nejlepší řešení lepší, test neuspěl.

- Test na horní mez 2 / test 3: Je dáno budované řešení A a prvek x_j, který se nevešel do batohu v souvislém intervalu prvků přidávaným do řešení. Je vypočítána horní mez řešení A stejným způsobem jako horní mez pro celou instanci batohu. Tato mez je následně porovnána s dosavadním nejlepším řešením. Pokud je dosavadní nejlepší řešení lepší, test neuspěl.

- Test na optimalitu / test 4: Řešení je porovnáno s dosavadním nejlepším řešením, pokud je hodnota jeho účelové funkce lepší, je jím nahrazeno dosavadní nejlepší řešení.

Algoritmus je rozdělen na tři části, v textu budou nazývané inicializační, dopředný krok a zpětný krok.

Inicializační část nastaví proměnné na počáteční hodnoty, seřadí prvky dle efektivity a nalezne horní mez celé instance problému. Pokud při výpočtu horní meze byla zaplněna celá kapacita batohu, je daná horní mez i optimálním řešením a algoritmus končí. Pokud se tak nestalo následuje dopředný krok.

Dopředný krok se skládá ze dvou částí, budovatelské a ukládací:

- Budovatelská část: Pokud se vpravo od posledního prvku přidaného do řešení nacházejí prvky s vahou menší než zbývající kapacita batohu jsou procházeny po jednom prvky napravo od prvku, který se v posledním dopředném chodu již nevešel do batohu. Každý tento prvek je podroben testu 2, pokud neuspěje pokračuje se zpětným krokem, jinak je takto postupováno dokud není nalezen první prvek, který lze umístit do batohu, aniž by jeho přidání do budovaného řešení porušilo omezující podmínku na kapacitu batohu. Následně je ve směru zleva doprava od nalezeného prvku přidán do batohu největší možný souvislý interval prvků jenž neporušuje omezující podmínku na kapacitu. Po tomto je proveden test 1, poté test 3, pokud neuspěje pokračuje se zpětným chodem, jinak algoritmus pokračuje do ukládací části.

- Ukládací část: Uloží právě budované řešení a pokud je možné přidat do batohu prvky napravo od naposled přidaného prvku, je zavolána znovu budovatelská část. Jinak je s právě zbudovaným řešením proveden test 4 a následuje zpětný krok.

Při zpětném kroku jsou prvky nahrazovány zprava, respektive od prvního prvku zprava zahrnutého v řešení, tedy od prvků s nejmenší efektivitou. Nechť je nahrazovaný prvek označen x. Tento prvek je odstraněn z řešení.

Jestliže před odstraněním prvku x zbývala kapacita na přidání prvků vpravo od x, potom algoritmus pokračuje dopředným krokem. Pokud před odstraněním prvku x nezbývala kapacita na přidání prvků vpravo od něj a zároveň prvek není poslední, stojí následující postup na úvaze, že řešení je možné zlepšit substitucí tohoto prvku za prvky vpravo od něj, tedy prvky s nižší efektivitou, jen pokud:

1. Je prvek x nahrazen jiným prvkem, který má vyšší hodnotu zisku a je možné ho umístit do řešení z hlediska zbývající kapacity. Tento prvek musí mít také větší váhu, neboť má nižší efektivitu než prvek x.

2. Je prvek x nahrazen alespoň dvěma prvky, každý s nižší vahou a s nižší hodnotou zisku, které ale mají společně vyšší hodnotu zisku než prvek x. Pokud by měl přidávaný prvek větší váhu a nižší hodnotu zisku, pak by řešení jednoznačně zhoršil, dále nemůže mít nižší váhu a vyšší hodnotu zisku, neboť potom by měl vyšší efektivitu a ležel nalevo od prvku x.

Algoritmus po jednom testuje prvky napravo od nahrazovaného prvku na zmíněné dva případy, dokud je test 2 pro tyto prvky úspěšný a zbývají prvky

k otestování, jinak následuje zpětný krok. Nechť nalezne prvek y. Pokud nastane případ 1 je proveden test 4 pro řešení s prvkem y, poté se testuje další prvek v pořadí na zmíněné dva případy. Pokud nastane případ 2, je pro daný prvek proveden test 2, tedy je testováno jestli by dosavadní nejlepší řešení bylo zlepšeno pokud by k prvku y byl nalezen prvek nebo prvky se stejnou efektivitou a s kterým/i by měli dohromady velikost rovnou zbývající kapacitě batohu, což je nejlepší možný případ, který může nastat. Pokud test 2 uspěje, pokračuje algoritmus dopředným krokem, který bude hledat další prvky k prvku y. Jinak je testován další prvek v pořadí na zmíněné dva případy.

Tímto postupem je dosaženo prohledávací strategie DFS, čímž je redukována paměťová náročnost, neboť algoritmus si musí pamatovat jen aktuální vrchol. Nicméně zanořování není slepé, ale mířené na nejnadějnější řešení pomocí přidávání prvků zleva, neboť díky řazení dle efektivity mají tyto prvky vyšší efektivitu a zároveň takovéto řešení bude dobře využitelné k prováděným testů využívající dosavadní nejlepší řešení. Zmíněný dopředný krok a zpětný krok reprezentují dopředný krok a zpětný krok algoritmu DFS. Prořezávání na základě mezí realizují jednotlivé výše zmíněné testy, dále algoritmus ve svém průběhu kontroluje omezující podmínku na kapacitu a nedochází tedy k prohledávání nesplnitelných řešení.

3.2 Algoritmus *DPwL*

Pro popis tohoto přístupu bude autor vycházet z [15] včetně uvedených pseudokódů.

3.2.1 Stavy a dominance stavů

Dle [8] stav batohu představuje nějaké řešení vyjádřené dvojicí {w,p}, kde w je součet vah a p součet hodnot zisků prvků umístěných v tomto řešení do batohu. Dále lze z dalšího prohledávání vyloučit stav, který má při stejné nebo vyšší váze menší hodnotu zisku oproti jinému stavu, poté je jím dominován. Navíc pro tento stav platí, že pokud je v nějakém řešení nahrazen stavem, který jej dominuje, vznikne lepší řešení a tedy dominovaný stav nikdy nemůže vyústit v optimum.

3.2.2 Průběh algoritmu

Tento algoritmus je založen na generování a slučování podle váhy vzestupně uspořádaných seznamů nedominujících se stavů. Nechť L_{j-1} představuje seznam se stavy získanými z množiny prvků $\{x_1, ..., x_{j-1}\}$, potom L_j je získán vygenerováním nového seznamu L'_{j-1} přičtením prvku $x_j = \{w_j, p_j\}$ ke každému stavu v seznamu L_{j-1} a sloučením L_{j-1} s L'_{j-1}, kdy toto sloučení zachovává zmíněné vlastnosti, tedy eliminací dominovaných stavů a uspořádanost.

Algoritmus 1 Algoritmus DPwL

Vstup: - $(w_1, ..., w_n), (p_1, ..., p_n)$
Výstup: stav s největším ziskem

1: $L_0 = \langle (0, 0) \rangle$
2: **for** $i = 1$ **to** n **do**
3: $L'_{j-1} := L_{j-1} \oplus (w_j, p_j)$ {přidej (w_j, p_j) ke každému stavu z L'_{j-1}}
4: vymaž všechny stavy z L'_{j-1}, které překračují kapacitu batohu
5: $L_j := merge(L_{j-1}, L'_{j-1})$
6: **return** stav s největším ziskem z L_n

Z uspořádanosti podle váhy a eliminace dominovaných stavů plyne také uspořádanost podle zisku. Neboť pokud je vzat stav A a libovolný stav B nacházející se v seznamu před ním by měl vyšší hodnotu zisku, potom by stav B, vzhledem ke své nižší váze a vyššímu zisku, dominoval stav A, což by byl spor s tím, že jsou eliminovány dominované stavy.

Při slučování dvou seznamů do nového seznamu je postupováno tak, že z L'_{j-1} *a* L_{j-1} je vybrán stav s nejmenší vahou, označme (w, p), a odstraněn z daného seznamu. Dále je tento stav porovnán se stavem s největší vahou z nového seznamu, označme (w', p'). Následně mohou nastat z hlediska dominance stavů tři možnosti, stav (w, p) je dominován stavem (w', p'), stav (w', p') je dominován stavem (w, p), mezi stavy (w, p) a (w', p') není z hlediska dominance žádný vztah.

Algoritmus 2 Sloučení dvou seznamů

Vstup: L_{j-1}, L'_{j-1}
Výstup: L_j

1: $L_j = \langle (\infty, 0) \rangle$
2: přidej na konec (L_{j-1} *a* L'_{j-1}) *stav* $(\infty, 0)$
3: **repeat**
4: vezmi stav (w, p) s nejmenší vahou z L_{j-1} *a* L'_{j-1} a odstraň ho z daného listu
5: **if** $w \neq \infty$ **then**
6: vezmi stav (w', p') s největší vahou z L_j
7: **if** $p > p'$ **then**
8: **if** $w \leq w'$ **then**
9: nahraď stav (w', p') v L_j stavem (w, p)
10: **else**
11: přidej stav (w, p) na konec L_j
12: **until** $w = \infty$
13: **return** L_j

Na rozdíl od základního přístupu k řešení batohu 0-1 pomocí *Dynamic programming by weights*, kdy je hledáno optimální řešení pro každou kapacitu, hledá tento přístup optimálního řešení jen pro kapacity rovné součtu vah nějaké kombinace prvků. Tento předpoklad neporuší nalezení optimálního řešení, neboť každé řešení má přiřazenou kapacitu, kterou zabírá, a jež je kombinací vah prvků v něm zahrnutých. Kapacitu, ke které nepatří žádné řešení není nutné uvažovat, neboť se neporuší podmínka zvážení všech možných řešení jako optimálních.

3.2.3 Složitost

Po sloučení dvou seznamů má výsledný seznam díky eliminaci dominovaných stavů velikost maximálně $c + 1$, kde c je kapacita batohu. Algoritmus provede n iterací, kde n je počet prvků, což dává asymptotickou složitost $\mathcal{O}(c{\cdot}n)$. Neboť stavy představují nějaké řešení, kterých je maximálně 2^n, lze asymptotickou složitost také vyjádřit jako $\mathcal{O}(min(c \cdot n, 2^n)$.

3.2.4 Dynamické programování s horní mezí

V [15] je popsán přístup dynamického programování kombinovaný s technikou *prořezávání* jako v metodě větví a mezí. Při použiti výše popsaného přístupu algoritmu *DPwL* lze na každý stav nahlížet jako na vrchol v metodě větví a mezí. Poté je pro tyto stavy nalezena horní mez a porovnána s největší dosud nalezenou hodnotou zisku nějakého stavu. Pokud je horní mez menší, je možné stav odstranit ze seznamu, neboť nevyústí v lepší řešení nežli daný stav s momentálně největší hodnotou zisku.

3.3 Algoritmus *HS1974*

Algoritmus je popsán v [19] a autor z tohoto zdroje bude v této sekci vycházet, pokud nebude uvedeno jinak.

Myšlenka, kterou algoritmus přináší je rozdělení množiny prvků na dvě stejně velké části, až na případný lichý prvek, které jsou brány jako nezávislé instance problému batohu. Tyto instance jsou vyřešeny ve smyslu, který bude uveden níže. Následně je v těchto dvou řešeních daných instancí nalezeno řešení originální instance.

3.3.1 Řešení dvou nezávislých instancí

Algoritmus požaduje pro nezávislé instance problému nalezení všech nedominujících se stavů z množin $\{x_1, ..., x_j\}$ pro $j = 1...n$, kde n je počet prvků v instanci. Tyto stavy musí být seřazeny dle váhy. Díky seřazenosti dle váhy a eliminaci dominovaných stavů jsou stavy seřazeny i dle zisku. Autor si pro implementaci vybral algoritmus *DPwL* zmíněný v kapitole 3.2.

3.3.2 Protisměrné prohledávání

Protisměrné prohledávání dvou podle váhy a zisku uspořádaných seznamů nedominujících se stavů spočívá v současném procházení těchto seznamů tak, že jeden ze seznamů je prohledáván od nejmenšího stavu a druhý od největšího z hlediska jejich váhy. Algoritmus využívá faktu, že díky zmíněným vlastnostem těchto seznamů platí pro dva takové, označme A a B, že pokud stav y z A je sloučen se stavem x z B vznikne stav p, potom žádný stav menší než y nebude po sloučení s x lepší než p a žádný stav menší než x nebude po sloučení s y lepší než p. Složitost protisměrného prohledávání je $\mathcal{O}(max(|A|, |B|))$.

Algoritmus 3 Protisměrné prohledávání dvou seznamů

Vstup: seznam A, seznam B
Výstup: Stav z A a stav z B, které mají dohromady nejvyšší hodnotu zisku

```
1: {A a B jsou indexovány od nuly}
2: a := |A|, b := |B|
3: indexToListB = b, indexToListA = 1
4: c := kapacita batohu
5: maximum = 0
6: for i = indexToListB − 1 to 0 do
7:   while indexToListA < a do
8:     if A_i(w) + B_j(w) > c then
9:       break
10:    else if A_i(p) + B_j(p) > maximum then
11:      maximum = A_i(p) + B_j(p)
12:    indexToListA = indexToListA + 1;
13: return maximum
```

3.3.3 Redukce složitosti

Nechť je dána instance problému batohu 0-1 o kapacitě c a n prvcích, poté je počet všech řešení 2^n. Pokud bude instance problému rozdělena na dvě části o $n/2$ prvcích, potom budou mít obě části počet všech řešení $2^{n/2}$, celkem tedy $2 \cdot 2^{n/2} = 2^{n/2+1}$. Tímto tedy dojde ke zmenšení počtu všech řešení o zhruba druhou odmocninu. Ze složitosti algoritmu *DPwL* a části *protisměrné prohledávání* plyne složitost celého algoritmu $\mathcal{O}(min(c \cdot n, 2^{n/2})$.

Kapitola 4

Paralelizace

V této kapitole bude podán stručný úvod do paralelizace a následně navrhnuty paralelní verze vybraných algoritmů popsaných v předchozí kapitole.

4.1 Stručný úvod do paralelizace

Autor pro tuto část vycházel z [21] pokud není uvedeno jinak.

Paralelizace představuje přizpůsobení algoritmů k zpracovávání více výpočetními jednotkami současně. Ideálním stavem je při použití n výpočetních jednotek n-krát rychlejší běh algoritmu, což často není možné kvůli částem algoritmu zpracovávaným sekvenčně. Maximální možné zrychleni při zvážení sekvenční a paralelizovatelné části algoritmu je definované Amhdalovým zákonem.

K ohodnocení kvality paralelizace sekvenčního algoritmu slouží různé metriky. Mezi hlavní metriky sloužící pro ohodnocení paralelního algoritmu patří:

1. **Zrychlení** porovnává čas nutný ke zpracování úlohy při jedné výpočetní jednotce ku více výpočetním jednotkám. Zrychlení je vyjádřeno vzorcem $S_P = T_1/T_P$, kde T_P vyjadřuje čas nutný ke zpracování úlohy při p výpočetních jednotkách. Ideální zrychlení je *lineární*, tedy při P výpočetních jednotkách je dosaženo P krát rychlejšího výpočtu. Některé speciální případy mohou dosahovat *superlineárního zrychlení*, například při paralelizaci algoritmu *větví a mezí* může dojít k odříznutí větve, díky hodnotě nalezené v jiné souběžně procházené větvi, kterou by jinak sekvenční algoritmus byl nucen projít.

2. **Efektivita** udává průměrný přínos jedné výpočetní jednotky na zrychlení. Je vyjádřena vzorcem S_p/P. Ideální hodnota je 1, což představuje lineární zrychlení.

Paralelizace z hlediska škálovatelnosti při rostoucím počtu dat lze rozdělit na dva přístupy:

- **Datový paralelismus** je přístup při kterém s rostoucím počtem dat roste možnost paralelizace, jedná se tedy o přistup s dobrou škálovatelností při rostoucím počtu dat.

- **Funkční paralelismus** označuje přistup, kdy jsou souběžně zpracovávány odlišné úlohy na stejných datech tak, že každou z těchto úloh zpracovává právě jedno výpočetní jádro. Z toho plyne, že výkonnost se zvedne o maximálně konstantní faktor rovný počtu takto zpracovávaných funkcí, pokud jsou tyto funkce vzájemně nezávislé a mají stejnou časovou náročnost. Tento přístup nemá vlastnost škálovatelnosti při rostoucím objemu dat ke zpracování.

Modely pro paralelní výpočet představují způsob implementace paralelizace, autor uveden jen ty, které budou vyžity v praktické části této práce:

1. **Work pool model:** V tomto modelu se dle [22] nachází uložiště pro úlohy určené ke zpracování ze kterého jsou dynamicky přiřazovány jednotlivým výpočetním jednotkám. Tyto úlohy mohou být generovány dynamicky v průběhu výpočtu, nebo mohou být vygenerovány staticky na začátku výpočtu.

2. **Iterační paralelismus:** Je dle [23] speciálním případem *datového paralelismu*, při kterém se souběžně vykonávají iterace cyklu. Paralelně vykonávané iterace musí být datově nezávislé.

Paralelizace byla implementována pomocí paralelního zpracování vlákny, v textu proto bude místo pojmu *výpočetní jednotka* uváděn pojem *vlákno.*

4.2 Návrh paralelizace algoritmu *MT1*

Vzhledem ke grafové interpretaci metody *větví a mezí* se jako přímočarý způsob paralelizace nabízí přiřazování částí stavového prostoru k prohledání jednotlivým vláknům. Při dělení problémů je třeba vyvážit zátěž co nejrovnoměrněji mezi výpočetní jednotky a zároveň udržovat režii spojenou s paralelním zpracováním v rozumných mezích. Pokud by úlohy byly děleny na příliš velké části, roste riziko, že nějaká z úloh bude příliš časově náročná a jiná zase příliš málo. Kdyby naopak úlohy byly děleny na příliš malé části, poté by sice bylo eliminováno riziko s nerovnoměrnou zátěží, ale režie spojená s přidělováním těchto úloh výpočetním jednotkám by mohla být příliš vysoká. Je tedy třeba vyvážit zmíněné dvě negativní skutečnosti.

Prohledávací strategie *DFS* může být implementována pomocí datové struktury *zásobník*, kdy vrcholy určené k průchodu zpětným krokem se vkládají na vrchol zásobníku. Poté dle [24] je prohledávací strategii *DFS* možné paralelizovat tak, že každé vlákno má svůj zásobník. Pokud vlákno projde zpětným krokem všechny vrcholy ve svém zásobníku, požádá jiné vlákno o přidělení

práce. Dané vlákno poté předá část obsahu svého zásobníku žádajícímu vláknu. Dále je třeba zvolit strategii k rozdělování zátěže mezi vlákna. Nejjednodušší strategií je náhodný výběr vlákna sdílející zátěž.

Nechť je dopředným krokem algoritmu nalezeno první řešení, toto řešení je v grafové reprezentaci *větví* a prvek za posledním přidaným je *kritický prvek*. Toto řešení by mělo být dobře využitelné k *prořezávání*, neboť prvky byly přidávány od prvku s největší efektivitou. Autor se rozhodl tuto *větev* dělit po malém počtu prvků určených k projití zpětným krokem mezi vlákna, poslední dostane zbývající nerozdělenou část. Tedy vlákna se budou snažit nahradit v tomto prvním řešení do něj zahrnuté prvky s nejmenší efektivitou, tím by mělo dojít k nalezení řešení dobře využitelného k prořezávání. Následně každé vlákno zpracuje jemu přiřazené vrcholy odděleně. Po dokončení počátečních úloh začne docházet k přerozdělování práce mezí vlákny.

Sdílenými prostředky mezi vlákny je dosavadní nejlepší řešení a proměnná signalizující nalezení optimálního řešení. Nesdílení dosavadního nejlepšího řešení mezi vlákny neporuší korektnost algoritmu, neboť důsledkem je potencionální prohledání větší části stavového prostoru, nežli by bylo nutné při jeho znalosti napříč vlákny. Tedy stojí proti sobě zvýšená režie a zbytečné procházení části stavového prostoru, ve kterém neleží optimální řešení. Autor se kloní spíše k co nejrychlejšímu sdílení mezi výpočetními jednotkami, aby nedocházelo k zbytečnému procházení stavového prostoru a tedy k plýtvání výpočetním výkonem vláken. Dále při úspěchu testu 1 uvedeného v kapitole popisující algoritmus *MT1*, tedy při nalezení optimálního řešení, je třeba signalizovat všem výpočetním jednotkám, že není třeba dále prohledávat stavový prostor. Tato signalizace by měla být také sdílena co nejrychleji, aby výpočet nepokračoval po nalezení optimálního řešení.

4.3 Návrh paralelizace algoritmu *HS1974*

Paralelizace části dynamického programování, kterou autor implementoval algoritmem *DPwL*, bude popsána v kapitole 4.4. Algoritmus *HS1974* rozdělí množinu prvků na dvě disjunktní části, které jsou až do závěrečného protisměrného prohledávání zpracovávány jako nezávislé instance batohu, tudíž je tyto části možné zpracovat paralelně.

Pro dosažení vyšší úrovně paralelizace se nabízí rozdělit množinu prvků na více než dvě disjunktní části, například na počet částí rovný počtu vláken podílejících se na výpočtu, nechť je počet vláken n. Každé vlákno následně vyřeší instanci problému tvořenou ji přidělenými prvky, tedy nalezne všechny nedominující stavy seřazené dle váhy. Pro nalezení optimálního řešení originální instance problému je dle [19] možné zvolit ze dvou přístupů:

1. Slučovat po dvojicích seznamy, dokud nezbudou dva, ve kterých je nalezeno optimum pomocí *protisměrného prohledávání*. Při sloučení dvou seznamů je třeba sloučit každý stav s každým, aby byly zváženy všechny

možné konfigurace řešení. Autorovi se nepodařilo navrhnout ani nalézt efektivní algoritmus stojící na tomto přístupu.

2. Neslučovat jednotlivé seznamy, ale hledat napříč jimi optimální řešení. Dle [19] je tato možnost při sekvenčním zpracování časově náročnější než-li dělení na 2 části. V [25] je uveden paralelní algoritmus stojící na této myšlence řešící problém batohu s asymptotickou časovou složitostí $\mathcal{O}(\log n \cdot \log c)$ při paralelizaci vyžadující $\mathcal{O}(n \cdot c^2/(\log n \cdot (\log c)^2))$ vláken, kde n je počet prvků a c je kapacita batohu, což i při malých instancích batohu vyžaduje desítky vláken. Z důvodu takto vysokých počtů potřebných vláken se autor implementací této paralelizace nebude zabývat. Sekvenční průběh je časově náročnější oproti algoritmu $HS1974$, což je v souladu s uvedeným tvrzením z [19].

Protisměrné prohledávání lze paralelizovat rozdělením seznamu přes který se iteruje vnějším cyklem na n částí. Dále pro každou část je určen index, na kterém začít iterovat ve vnitřním cyklu, aby nebyly vykonávány nadbytečné iterace. Tento index je určen pro každou část, až na poslední, tak, že pro rozdíl kapacity batohu a váhy nejmenšího stavu v následující části je hledána horní mez podle váhy v seznamu přes který se iteruje ve vnitřním cyklu.

4.4 Návrh paralelizace algoritmu *DPwL*

Pro tento algoritmus autor navrhl následující paralelizaci:

1. Paralelizovat lze přidávání prvku ke každému stavu v seznamu (viz řádky 3 a 4 v pseudokódu) pomocí rozdělení seznamu na části. Tyto části jsou nezávislé a mohou být zpracovávány paralelně.

2. Dva seznamy ve funkci *merge* by bylo také možné rozdělit na části a ty slučovat paralelně, bohužel nastává problém s tím, že jednotlivé části nejsou nezávislé. Každá část se skládá z části prvního seznamu a části druhého seznamu, které mají být sloučeny, ale každý stav z prvního seznamu musí být porovnán s rovným nebo největším menším z hlediska váhy z druhého seznamu a naopak, kvůli eliminaci dominovaných stavů.

   Autor se rozhodl pro následující způsob dělení seznamů na nezávislé části. Nechť jsou dva slučované seznamy označeny jako L_1 a L_2. V seznamu L_1 je vybrán stav, označme y, určující začátek jedné části, označme tuto část A. V L_2 je nalezen největší menší stav z hlediska váhy ke stavu y, označme x, dále první stav po stavu x, označme z, je začátek části z L_2 k části A. Následně mohou nastat 2 možnosti:

   a) Stavy y a z jsou si rovny z hlediska váhy.
   b) Stav z je větší než stav y z hlediska váhy, respektive je větší než po sobě jdoucí stavy $y_1, ..., y_j$, kde y_1 je y.

Pokud nastane případ a) jsou tyto části nezávislé na předchozích. Pokud nastane případ b) je třeba stavem x ověřit dominanci nad stavy $y_1, ..., y_j$. Toto lze implementovat tak, že dominované stavy jsou přesunuty do předcházející části, kde budou odstraněny, neboť se v ní nachází stav x, který je dominuje.

Tedy algoritmus bude dělit seznam L_1 na části pro jejichž stav s nejmenší vahou je hledán největší menší stav v druhém seznamu a ověřována dominance. Pokud se počet stavů v nějaké části příliš odchýlí od ideálního rozdělení stavů na n stejných částí, kde n je požadovaný počet částí, algoritmus posune začátek dané části ze seznamu L_1. Počet úprav je omezen parametrem.

Algoritmus 4 Dělení dvou slučovaných seznamů na n částí

Vstup: L_1 := původní seznam L_2 := původní seznam s přidaným prvkem
Výstup: dělicí body pro seznam L_1 a L_2

1: p := kolikrát vyrovnávat velikost částí
2: $x = 0$
3: $y = 0$
4: **for** $i = 1$ **to** n **do**
5: $m := \frac{\text{zbývající nerozdělená část}}{\text{počet zbývajících dělení}}$
6: $z =$ stav v L_1 na pozici $x + m$
7: **for** $j = 1$ **to** p **do**
8: $s =$ největší menší stav v seznamu 2 ke stavu z
9: $t = z - x + s - y$ {počet stavů mezi x a z a mezi y a s včetně}
10: **if** t je příliš malé oproti $\frac{\text{počet všech stavů v listu 1 a 2}}{n}$ **then**
11: posuň z doprava
12: **else if** t je příliš velké oproti $\frac{\text{počet všech stavů v listu 1 a 2}}{n}$ **then**
13: posuň z doleva
14: **else**
15: break
16: $x = z$
17: $y = s$
18: přidej z a s jako dělící body
19: uprav části z hlediska dominance
20: **return** dělicí body

4.5 Návrh paralelizace metody *hrubé síly*

Pro paralelizaci lze vygenerovat n počátečních kombinací, kde n je počet vláken. Tyto kombinace budou dále rozvíjeny jednotlivými vlákny. Počáteční kombinace jsou generovány tak, že se nějaký prvek umístí nebo neumístí do batohu. Při použití jednoho prvku pro vygenerování počátečních kombi-

nací vzniknou 2 kombinace, pokud se použijí 2 prvky vzniknou 4 kombinace. Náročnost jednotlivých počátečních kombinací se liší podle kapacity zabrané počáteční kombinací.

Kapitola 5

Implementace

V této kapitole bude popsán výběr technologií a implementace jednotlivých algoritmů.

5.1 Výběr technologií

Implementace byly realizovány v programovacím jazyku *C++* a k paralelizaci bylo využita, dle zadání, technologie *OpenMP*. Nejdříve byly implementovány sekvenční verze, následně na nich byly postaveny verze paralelní vycházející z navržených paralelizací uvedených v kapitole 4.

OpenMP

Pro popis technologie *OpenMP* autor vycházel z [23]. *OpenMP* je vysokoúrovňové API sloužící k programování vícevláknových aplikací nad sdílenou pamětí. Obsahuje 3 základní části:

1. direktivy pro kompilátor,
2. proměnné *OpenMP* prostředí,
3. knihovnu funkcí běhového prostředí *OpenMP*.

OpenMP pracuje s jedním hlavním(*master*) vláknem. V explicitně určených regionech(*parallel regions*) jsou zapojena další vlákna pomocí $fork-join$ mechanismu. Může se využívat takzvaného *thread pool*, kdy jsou využívána již existující vlákna, díky čemuž je zabráněno neustálému vytváření a zániku vláken. *OpenMP* podporuje datový i funkční paralelismus.

5.2 Implementace algoritmu *MT1*

V implementaci je třída *Item* reprezentující prvky přidávané do batohu. Implementace je rozdělena do několika funkcí, které realizují jednotlivé části algoritmu:

1. Funkce *init* realizuje iniciální část algoritmu. K seřazení prvků byla využita funkce *sort* z knihovny *std*.

2. Funkce *loop* realizuje cyklus ve kterém je střídavě volán dopředný a zpětný krok dokud nenastane některá z podmínek pro konec algoritmu.

3. Funkce *forwardMove* realizuje budovatelskou část dopředného kroku.

4. Funkce *saveLocalSolution* realizuje ukládací část dopředného kroku.

5. Funkce *saveGlobalSolution* vykonává test 4, tedy porovná aktuálně nalezené řešení s dosavadním nejlepším řešením, které případně nahradí. Tato metoda je poslední fází dopředného kroku, poté se algoritmus vrátí do metody *loop*, kde je zavolána funkce *backTrackMove*.

6. Funkce *selectItemForBacktrackMove* nalezne první prvek zprava, který je zahrnut v řešení.

7. Funkce *backTrackMove* zavolá metodu *selectItemForBacktrackMove*, která určí prvek k odstranění z řešení. Následně tato funkce realizuje zpětný krok algoritmu.

8. Funkce *substitute* realizuje část algoritmu, která se snaží substituovat za poslední přidaný prvek prvky napravo od něj.

9. Funkce *compute UB* spočítá horní mez.

Algoritmus byl paralelizován způsobem popsaným v části analyzující paralelizaci, s využitím modelu *Work pool* , který byl implementován pomocí *OpenMP* direktivy *pragma omp task*. K reprezentaci zásobníku bylo použito pole, neboť díky prohledávací strategii DFS je největší možné zanoření rovno počtu prvků a je tedy možno ho alokovat bez nutnosti budoucího zvětšování velikosti.

Pro aktualizaci hodnoty sdíleného nejlepšího řešení byl použit mechanismus, že vlákno podílející se na výpočtu po určitém počtu dopředných a zpětných kroků aktualizuje svoji lokální hodnotu sdíleného nejlepšího řešení pokud je horší, pokud ne, nahradí ho svým nejlepším řešením, toto se děje v kritické sekci. Tento počet je třeba nastavit dle výkonu procesoru, aby nedocházelo k příliš častému přístupu do kritické sekce.

Sdílení práce mezi vlákny bylo realizováno společným čítačem vyjadřujícím počet vláken čekajících na přidělení práce. Pokud vlákno dokončí přidělenou úlohu inkrementuje čítač. Práce s čítačem se děje v kritické sekci. Při aktualizaci hodnoty sdíleného nejlepšího řešení se zároveň vlákno podívá, jestli nějaké jiné vlákno nepožaduje přidělení práce, pokud ano, dekrementuje čítač a přidá pomocí *OpenMP* direktivy *pragma omp task* jako úlohu ke zpracování polovinu svého zásobníku. Tedy práci bude sdílet náhodné vlákno.

Funkce přidané v paralelní verzi:

1. Funkce *searchProvidedNodes* nahradila funkci *loop*. Tato funkce také vykonává cyklus dopředných a zpětných kroků a navíc je v ni obsaženo popsané sdílení práce a aktualizace sdíleného nejlepšího řešení.

2. Funkce *initialMove* vygeneruje první řešení a rozdělí ho mezi vlákna.

3. Funkce *updateValuesAfterSharingWork* aktualizuje lokální proměnné vlákna sdílející práci.

4. Funkce *saveGlobalSolutionGlobal* se pokusí uložit lokální nejlepší řešení vlákna jako sdílené nejlepší řešení.

5. Funkce *requestWork* inkrementuje čítač vyjadřující počet vláken bez práce.

6. Funkce *provideWork* dekrementuje čítač pokud je větší než 0 a signalizuje jestli existuje vlákno čekající na přidělení práce.

5.3 Implementace algoritmu *HS1974* a *DPwL*

Implementace algoritmů *HS1974* a *DWwL* je totožná až na následující rozdíly. Pro algoritmus *HS1974* je originální instance rozdělena na dvě menší, každá o polovině prvků, a pro nalezení optimálního řešení je využito *protisměrné vyhledávání*, dále u *HS1974* jsou ve stavech ukládány prvky v nich zahrnuté pro nalezení konfigurace optimálního řešení.

Pro reprezentaci seznamů se nabízí implementace pomocí datové struktury *pole* nebo datové struktury *spojový seznam*. Autor se rozhodl pro pole, neboť algoritmus prochází stavy v seznamu sekvenčně a díky kontinuálnímu uložení v paměti se do výpočetního času algoritmu promítne zrychlení v důsledku nahrání sousedních stavů do *cache* paměti procesoru. Pro reprezentaci pole byl využit kontejner *vector* z knihovny *std*. V průběhu algoritmu *DPwL* jsou potřeba 3 seznamy, pro algoritmus *HS1974* je potřeba ještě jeden navíc, neboť jsou počítány dvě menší instance problému místo jedné. Aby nedocházelo realokacím paměti, které jsou časově náročné, je na začátku výpočtu pro kontejner *vector* rezervováno místo v pamětí pro počet stavů rovný kapacitě batohu zvýšené o jedna, což je maximální možný počet.

Pro nalezení výsledné konfigurace optimálního řešení je pro každý stav třeba uchovávat prvky v něm zahrnuté. Pro ukládání konfigurace řešení bylo použito bitové pole implementované pomocí kontejneru *vector* s argumentem *bool*, který optimalizuje svoji velikost tak, že každý prvek v poli alokovaném tímto kontejnerem zabírá jeden bit.

V implementaci je třída *Item* reprezentující prvky přidávané do batohu a třída *State* reprezentující stavy s seznamech. Implementace je rozdělena do několika funkcí, které realizují jednotlivé části algoritmu:

1. Funkce *computeLists* vykonává cyklus z algoritmu *DPwL* voláním funkcí *addItemToList* a *mergeTwoListsToNewOne* uvedených níže. U algoritmu *HS*1974 jsou prvky z instance problému rozděleny na dvě části a cyklus se provede dvakrát pro dvě menší instance problému.

2. Funkce *addItemToList* implementuje řádky 3 a 4 algoritmu *DPwL*. Tedy vytvoří seznam z jiného seznamu přidáním hodnoty zisku a váhy určeného prvku ke všem stavům tohoto seznamu.

3. Funkce *mergeTwoListsToNewOne* implementuje metodu *merge*. Tedy sloučí dva seznamy do jednoho s vyloučením dominovaných stavů.

4. Funkce *findOptimum* implementuje hledání optimálního řešení ve dvou seznamech způsobem uvedeným v popisu metody *Protisměrné prohledávání*. Tato funkce je využita jen v algoritmu *HS*1974.

Při implementaci paralelizace zpracování dvou nezávislých instancí u algoritmu *HS*1974 se ukázalo, že v kombinaci s paralelizací algoritmu *DPwL* bylo nutné použít vnořené(*nested*) paralelizace, která při použití technologie *OpenMP* přinášela zvýšenou režii a v důsledku toho značně horší výsledky nežli verze s pouze paralelním *DPwL* nebo pouze paralelním zpracováním dvou nezávislých instancí. Bylo tedy možné využít jen jednu z těchto paralelizací. Autor se rozhodl pro paralelizaci *DPwL* neboť u ní není omezen počet vláken podílejících se na výpočtu na dvě, jako u paralelního zpracování dvou nezávislých instancí.

Byla tedy implementována paralelizace algoritmu *DPwL* a u algoritmu *HS*1974 ještě paralelizace protisměrného prohledávání.

V paralelní verzi přibyly následující funkce:

1. Funkce *computeDivisionOfLists* před vykonáním metody *merge* rozdělí seznamy na paralelně zpracovávané části. Funkce se snaží rozdělit seznamy na n částí, kde n je zadaný parametr a měl by odpovídat počtu vláken podílejících se na výpočtu.

2. Funkce *checkDominance* při rozdělení seznamů na paralelně zpracovávané části testuje dominanci stavů mezi částmi, a případně přesune stavy do jiné části, jak bylo popsáno v sekci pojednávající o paralelizaci algoritmu *DPwL*.

3. Funkce *removeBlankSpaces* je volána na poslední vytvořený seznam, aby odstranila prázdná místa, viz níže, po paralelním zpracování.

Metody ze sekvenční verze byly paralelizovány následovně:

1. Ve funkci *mergeTwoListsToNewOne* je zavolána funkce *computeDivisionOfLists*, následně jsou paralelně zpracovány části seznamů určené zavolanou funkcí. Pro každou část je určeno, kam začít vkládat stavy do

nového seznamu. Tyto části jsou určeny součtem stavů zpracovávaných v dané části. Neboť v každé této paralelně zpracované části může nastat dominance stavů a skutečný počet být nižší nežli součet, zůstane v takovém případě po dané části v novém seznamu prázdné místo. Toto místo je zaznamenáno, dále s ním pracuje funkce *addItemToList*.

2. Funkce *addItemToList* zpracuje paralelně n sloučených částí z funkce *mergeTwoListsToNewOne*, kde n je počet částí, na které byl seznam rozdělen. Dále jsou ve sloučeném seznamu odstraněny prázdná místa mezi paralelně zpracovanými částmi.

3. Funkce *protismerneVyhledavani* je paralelizována rozdělením seznamu přes který iteruje vnější cyklus *for* na n rovnoměrných částí, kde n je počet vláken podílejících se na výpočtu. V druhém seznamu je k těmto částem nalezen odpovídající prvek od kterého začít iterovat. Následovně jsou tyto části prováděny nezávisle. Pomocí *OpenMP redukce* je nalezeno optimální řešení.

5.4 Implementace metody *hrubé síly*

Sekvenční metoda byla implementována způsobem zmíněným v kapitole 2.2.5.1. Autor se inspiroval algoritmem uvedeným v [26], kde je popsán pod názvem *Alternate Solution*.

Funkce v sekvenční verzi:

- Funkce *search* realizuje prohledání řešení pro danou počáteční kombinaci.

Paralelizace byla implementována dle analýzy paralelizace. Byl využit model *Work pool* implementovaný pomocí *OpenMP* direktivy *pragma omp task*. Hlavní vlákno vygeneruje počáteční kombinace, které budou následně paralelně zpracovány funkcí *search*. Na závěr jsou porovnány lokální optima jednotlivých vláken ze kterých je vybráno globální optimum.

Funkce přidané v paralelní verzi:

- Funkce *generateTasks* vygeneruje počáteční kombinace jako úlohy pomocí *OpemMP* direktivy *pragma omp task*.

Kapitola 6

Testování

V této kapitole budou otestovány autorovy implementace a konkurenční implementace na vybraných datových sadách a zhodnocena jejich výkonnost.

6.1 Konkurenční implementace

Autor vyhledal volně dostupné konkurenční implementace řešící problém batohu 0-1.

Algoritmus *minknap*

Algoritmus *minknap* zmíněný v kapitole 2.2. Implementace v jazyce C je dostupná z [27]. Implementace je sekvenční.

Algoritmus *combo*

Algoritmus *combo* zmíněný v kapitole 2.2 jako nejefektivnější algoritmus pro problém batohu 0-1. Implementace v jazyce C je dostupná z [27]. Implementace je sekvenční.

Konkurenční implementace algoritmu *větví a mezí*

Tento algoritmus je dostupný z [28]. Pro svou malou efektivitu už na nejmenších testovaných instancích ho autor z testování vyřadil.

Algoritmus *Dynamic programming by weights*

Tento algoritmus byl popsán v kapitole 2.2.3.1. Jeho implementace je dostupná z [29]. Autor se na tento algoritmus bude dále odkazovat pod zkratkou *DP1*. Dále implementace verze tohoto algoritmu poskytující pouze hodnotu zisku optimálního řešení, která je paměťově méně náročná, je dostupná z [30]. Autor se tento algoritmus bude odkazovat pod zkratkou *DP2*. Implementace jsou sekvenční.

6.2 Metodika testování

Měření času

Čas byl měřen po načtení vstupních dat do paměti, tedy od začátku výpočtu po vrácení optimálního řešení. K měření u autorových implementací a algoritmů $DP1$ a $DP2$ byla využita knihovna *chrono* ze standardní knihovny jazyka C++. Pro měření času u algoritmů *combo* a *minknap* byla použita knihovna *time.h* ze standardní knihovny jazyka C.

Hardware

Algoritmy byly testovány na procesoru *intel i5-3550 3.30 GHz* disponujícím 4 plnohodnotnými jádry. Dále 8 GB operační paměti, pro výpočet dostupných 6,5 GB.

Kompilace

Ke kompilaci autorových implementací a algoritmů $DP1$ a $DP2$ byl použit kompilátor *g++* ve verzi 7.4.0. Pro kompilaci konkurenční implementace algoritmu *minknap* a *combo* byl použit kompilátor *gcc* ve verzi 7.4.0. Dále byl u všech implementací použit optimalizující přepínač *-O3* a pro paralelní verze přepínač *-fopenmp*.

Typy dat

K testování byly použity datové sady dostupné z [27]. Z daného zdroje je dostupných 16 druhů datových sad. Autor si vybral 3, které považuje za nejvíce obecné. V těchto datových sadách jsou váhy a zisky prvků voleny z uniformního rozdělení, tedy každý prvek z daného intervalu má stejnou pravděpodobnost zvolení. Autor si k testování vybral následující datové sady:

- **Nekorelované:** Váha a zisk prvků jsou voleny nezávisle na sobě z intervalu $[1, R]$, kde R je zvolený parametr. Tedy není mezi nimi významná míra korelace.
- **Slabě korelované:** Váhy prvků jsou voleny z intervalu $[1, R]$, kde R je zvolený parametr. Zisk prvku x_j je volen z intervalu $[w_j - R/10, w_j + R/10]$.
- **Silně korelované:** Váhy prvků jsou voleny z intervalu $[1, R]$, kde R je zvolený parametr a $p_j = w_j + R/10$.

6.3 Výsledky měření

Měření bylo provedeno pro výše uvedené typy datových sad pro koeficienty R v hodnotách 1000 a 100000. Pro každý typ byl změřen čas pro 100 instancí problému daného typu a výsledné časy zprůměrovány. V tabulkách jsou uvedeny zprůměrované časy, v závorce je uveden maximální čas. Časy jsou uváděny v milisekundách. Písmeno n značí velikost instance problému, respektive počet

prvků. Písmeno R značí číselný rozsah pro váhy prvků. Dále u paralelních algoritmů značení [p] znamená, že bylo použito p vláken. V tabulkách pro sekvenční implementace je kvůli úspoře místa použito značení $U.$ pro nekorelovaná data, $W.\ C.$ pro slabě korelovaná data a $S.\ C.$ pro silně korelovaná data.

Tabulka 6.1: Naměřené časy pro sekvenční implementaci algoritmu *MT1*

MT1 - sekvenční verze							
n;R	U.	W. C.	S. C.	n;R	U.	W. C.	S. C.
50;1000	0(0)	0(0)	1,45(82)	50;100000	0(0)	0(0)	1,41(56)
100;1000	0(0)	0(0)	661,19(56073)	100;100000	0(0)	0(0)	1514,49(123491)
200;1000	0(0)	0(0)	-	200;100000	0(0)	0(0)	-
500;1000	0(0)	0(0)	-	500;100000	0(0)	0(0)	-
1000;1000	0(0)	0(0)	-	1000;100000	0(0)	0.02(1)	-
2000;1000	0,42(1)	0,47(4)	-	2000;100000	0,42(2)	0.69(2)	-
5000;1000	4,59(11)	6,39(271)	-	5000;100000	4,77(14)	4.84(16)	-
10000;1000	11,94(46)	34,81(1133)	-	10000;100000	19,78(45)	17.17(40)	-

Tabulka 6.2: Naměřené časy pro paralelní implementaci algoritmu *MT1*

MT1 - paralelní verze							
Nekorelované							
n;R / vlákna	[1]	[2]	[4]	n;R /vlákna	[1]	[2]	[4]
50;1000	0(0)	0(0)	0(0)	50;100000	0(0)	0(0)	0(0)
100;1000	0(0)	0(0)	0(0)	100;100000	0(0)	0(0)	0(0)
200;1000	0(0)	0(0)	0(0)	200;100000	0(0)	0(0)	0.01(1)
500;1000	0(0)	0(0)	0(0)	500;100000	0(0)	0(0)	0(0)
1000;1000	0,02(1)	0,23(1)	0,05(1)	1000;100000	0,01(1)	0,3(2)	0,14(2)
2000;1000	0,92(3)	1,1(2)	0,32(1)	2000;100000	0,96(3)	1,02(2)	0,5(3)
5000;1000	4,89(12)	4,43(18)	2,34(6)	5000;100000	6,01(15)	4,54(13)	2,74(6)
10000;1000	13,08(47)	9,04(36)	6,0(23)	10000;100000	20,84(47)	13,64(46)	8,07(21)
Slabě korelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	0(0)	0(0)	0(0)	50;100000	0(0)	0(0)	0(0)
100;1000	0(0)	0(0)	0(0)	100;100000	0(0)	0(0)	0.01(1)
200;1000	0(0)	0(0)	0,01(1)	200;100000	0(0)	0(0)	0()
500;1000	0(0)	0(0)	0,13(1)	500;100000	0(0)	0(0)	0(0)
1000;1000	0,13(1)	0,6(2)	0,41(2)	1000;100000	0,69(2)	0,57(1)	0,12(1)
2000;1000	0,97(3)	1,6(7)	0,69(3)	2000;100000	1,31(4)	1,58(4)	1,09(3)
5000;1000	7,19(272)	6,39(218)	3,34(108)	5000;100000	6,2(17)	6,75(19)	3,35(14)
10000;1000	31,2(972)	26,36(961)	9,42(371)	10000;100000	18,01(41)	23,54(94)	7,65(16)
Silně korelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	1,58(116)	1,41(97)	1,28(46)	50;100000	1,6(61)	1,42(53)	1,34(47)
100;1000	593,11(24071)	534,59(21462)	428,24(17600)	50;100000	1620,9(131676)	1484,36(120160)	1236,9(100293)
200;1000	-	-	-	200;100000	-	-	-
500;1000	-	-	-	500;100000	-	-	-
1000;1000	-	-	-	1000;100000	-	-	-
2000;1000	-	-	-	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

Tabulka 6.3: Naměřené časy pro sekvenční implementace algoritmu *HS1974* s konfigurací řešení

HS1974 (s konfigurací řešení) - sekvenční verze							
n;R	U.	W. C.	S. C.	n;R	U.	W. C.	S. C.
50;1000	1,5(3)	6,33(12)	29,75(73)	50;100000	1,33(4)	6,47(12)	80,69(595)
100;1000	10,1(16)	45,53(87)	335,42(535)	100;100000	9,73(16)	50,21(98)	5295,27(15076)
200;1000	41,74(66)	224,48(374)	1316,66(1819)	200;100000	42,21(70)	263,13(551)	-
500;1000	1152,2(1713)	5482,24(8219)	-	500;100000	1246,84(2143)	6857,75(12005)	-
1000;1000	-	-	-	1000;100000	-	-	-
2000;1000	-	-	-	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

Tabulka 6.4: Naměřené časy pro paralelní implementaci algoritmu *HS1974* s konfigurací řešení

HS1974 (s konfigurací řešení) - paralelní verze							
Nekorelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	1,91(3)	2,4(5)	3,13(4)	50;100000	1,9(3)	2,35(4)	3,23(9)
100;1000	9,21(14)	8,21(11)	9,24(19)	100;100000	8,91(13)	8,18(11)	9,01(12)
200;1000	24,78(36)	20,68(29)	21,54(31)	200;100000	25,34(39)	20,79(29)	21,54(29)
500;1000	712,66(1026)	400,18(570)	277,37(626)	500;100000	767,77(1227)	431,1(677)	288,82(440)
1000;1000	-	-	-	1000;100000	-	-	-
2000;1000	-	-	-	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-
Slabě korelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	7,3(13)	6,38(11)	6,67(12)	50;100000	7,38(14)	6,54(13)	6,8(13)
100;1000	38,24(75)	27,54(54)	25,3(52)	100;100000	42,47(82)	30,45(56)	28,48(57)
200;1000	111,09(175)	75,95(123)	68,58(108)	200;100000	128,04(265)	89,58(185)	78,98(155)
500;1000	3224,4(4778)	1753,24(2587)	1099,5(1593)	500;100000	4487,99(7887)	2443,12(4317)	1532,65(2726)
1000;1000	-	-	-	1000;100000	-	-	-
2000;1000	-	-	-	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-
Silně korelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	37,75(85)	31,9(73)	30,9(68)	50;100000	122,97(985)	106,19(852)	98,85(782)
100;1000	244,52(377)	167,98(260)	143,58(233)	100;100000	3968,63(10765)	2563,4(6777)	1680,3(4861)
200;1000	554,35(770)	368,56(514)	312,13(490)	200;100000	-	-	-
500;1000	-	-	-	500;100000	-	-	-
1000;1000	-	-	-	1000;100000	-	-	-
2000;1000	-	-	-	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

Tabulka 6.5: Naměřené časy pro sekvenční implementaci algoritmu *HS1974* bez konfigurace řešení

HS1974 (bez konfigurace řešení) - sekvenční verze							
n;R	U.	W. C.	S. C.	n;R	U.	W. C.	S. C.
50;1000	0(0)	0(0)	0,05(1)	50;100000	0(0)	0(0)	1(1)
100;1000	0(0)	0,78(1)	4,61(7)	100;100000	(0)	0,89(2)	134,42(390)
200;1000	1,12(2)	7,21(11)	32,03(48)	200;100000	1,07(2)	8,68(18)	2058,8(3443)
500;1000	20,44(29)	98,07(150)	264,85(350)	500;100000	21,84(35)	149,1(285)	-
1000;1000	149,48(212)	650,32(1002)	1189,65(1537)	1000;100000	174,29(258)	1237,36(2485)	-
2000;1000	1111,14(1446)	3767,97(5660)	5031,86(6492)	2000;100000	1418,46(1950)	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

Tabulka 6.6: Naměřené časy pro paralelní implementaci algoritmu *HS1974* bez konfigurace řešení

HS1974 (bez konfigurace řešení) - paralelní verze							
Nekorelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	1(1)	1,37(3)	2,1(6)	50;100000	1(1)	1,15(2)	2,11(9)
100;1000	2(2)	3,26(6)	4,07(6)	100;100000	2,05(4)	3,14(10)	4,3(13)
200;1000	5,36(6)	8,04(16)	9,17(20)	200;100000	5,26(6)	7,27(9)	9,1(14)
500;1000	29,21(37)	27,5(33)	28,54(32)	500;100000	30,81(43)	28,09(35)	28,33(36)
1000;1000	148,04(191)	100,9(120)	83,86(104)	1000;100000	170,55(239)	112,2(147)	88,23(109)
2000;1000	873,59(1089)	495,72(609)	336,12(588)	2000;100000	1136,58(1498)	629,91(811)	403,02(505)
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-
Slabě korelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	1(1)	1,72(5)	2,15(6)	50;100000	1(1)	1,16(5)	2,02(4)
100;1000	2,91(5)	4,27(7)	4,33(11)	100;100000	2,86(4)	3,98(8)	4,16(7)
200;1000	11,26(15)	10,60(14)	11,02(15)	200;100000	12,14(21)	11,28(15)	11,08(14)
500;1000	90,20(126)	59,57(79)	47,75(64)	500;100000	130,87(219)	80,69(126)	58,7(96)
1000;1000	485,98(700)	274,52(388)	184,93(255)	1000;100000	927(1749)	501,37(946)	311,82(583)
2000;1000	2410,76(3448)	1305,59(1878)	798,13(1183)	2000;100000	6990,32(12361)	3748,72(6723)	2373,57(4406)
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-
Silně korelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	1,37(2)	2,07(4)	2,18(6)	50;100000	2,52(15)	2,57(14)	2,63(11)
100;1000	6,55(9)	5,88(8)	5,84(11)	100;100000	112,8(301)	79,42(211)	67,26(186)
200;1000	27,52(36)	20,2(26)	17,34(24)	200;100000	1338,49(2174)	857,53(1407)	711,07(1176)
500;1000	163,59(206)	100,69(126)	73,42(93)	500;100000	-	-	-
1000;1000	631,76(811)	349,99(428)	233,92(284)	1000;100000	-	-	-
2000;1000	2508,14(3078)	1370,44(1794)	982,10(1704)	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

Tabulka 6.7: Naměřené časy pro sekvenční implementaci algoritmu *DPwL* bez konfigurace řešení

DPwL - sekvenční verze							
n;R	U.	W. C.	S. C.	n;R	U.	W. C.	S. C.
50;1000	0(0)	0,04(1)	1,58(4)	50;100000	0(0)	0,03(1)	45,73(312)
100;1000	0,07(1)	2,55(7)	12,61(26)	100;100000	0,13(1)	3,15(10)	754,67(2084)
200;1000	4,12(9)	19,72(54)	66,92(117)	200;100000	4,5(9)	28,53(89)	5090.62(10617)
500;1000	63,8(131)	257,09(558)	505,17(862)	500;100000	75,26(144)	498,97(1358)	-
1000;1000	485,12(858)	1533,3(3006)	2102,14(3615)	1000;100000	641,72(1276)	4057,45(10384)	-
2000;1000	3349,16(5178)	7740,06(13163)	7955,42(14291)	2000;100000	5264,74(9189)	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

Tabulka 6.8: Naměřené časy pro paralelní implementaci algoritmu *DPwL* bez konfigurace řešení

DPwL - paralelní verze							
Nekorelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	1(1)	1,29(3)	2,19(6)	50;100000	1(1)	1,3(5)	2,08(5)
100;1000	2,35(3)	3,46(5)	4,21(7)	100;100000	2,2(4)	3,19(6)	4,09(7)
200;1000	8,36(12)	9,05(11)	10,11(17)	200;100000	8,33(11)	8,94(11)	9,88(17)
500;1000	63,82(107)	45,27(72)	38,56(55)	500;100000	72,53(119)	49,44(71)	40,11(55)
1000;1000	365,08(574)	214,03(323)	144,63(214)	1000;100000	483,67(863)	270,86(462)	173,97(279)
2000;1000	2151,09(3284)	1148,86(1667)	704,04(1422)	2000;100000	3477,92(5772)	1824,51(3049)	1087,9(1871)
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-
Slabě korelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	1,03(2)	1,75(5)	2,17(6)	50;100000	1,04(2)	1,78(2)	2,02(3)
100;1000	4,54(9)	4,9(7)	5,16(11)	100;100000	5,12(12)	5,03(10)	5,13(11)
200;1000	21,29(45)	15,98(31)	13,92(21)	200;100000	28,27(69)	19,35(40)	15,74(29)
500;1000	189,3(378)	110,63(209)	75,22(134)	500;100000	359,03(865)	197,7(459)	122,86(275)
1000;1000	946,89(1711)	514,21(924)	319,21(555)	1000;100000	2724,9(6756)	1454,16(3657)	910,03(2390)
2000;1000	4393,55(7104)	2414,0(3990)	1615,87(2844)	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-
Silně korelované							
n;R / vlákna	[1]	[2]	[4]	n;R / vlákna	[1]	[2]	[4]
50;1000	2,43(5)	2,46(4)	2,44(3)	50;100000	36,83(231)	26,04(160)	21,91(143)
100;1000	10,85(18)	8,29(13)	7,26(10)	100;100000	478,88(1174)	304,91(737)	249,07(630)
200;1000	42,87(71)	28,07(41)	22,51(45)	200;100000	3006,13(5460)	1853,1(3355)	1548,38(2867)
500;1000	253,02(394)	147,18(303)	97,32(155)	500;100000	-	-	-
1000;1000	995,03(1522)	545,15(982)	368,25(678)	1000;100000	-	-	-
2000;1000	4005,18(6135)	2273,42(3776)	1862,23(3121)	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

Tabulka 6.9: Naměřené časy pro implementaci algoritmu *minknap*

minknap(konkurenční implementace)							
n;R	U.	W. C.	S. C.	n;R	U.	W. C.	S. C.
50;1000	0(0)	0(0)	0(0)	50;100000	0(0)	0,15(15)	2,33(16)
100;1000	0(0)	0(0)	0,6(15)	100;100000	0(0)	0,15(15)	11,23(108)
200;1000	0(0)	0,15(15)	1,05(15)	200;100000	0,15(15)	0,15(15)	48,35(281)
500;1000	0(0)	0,15(15)	5,4(15)	500;100000	0,3(15)	0,3(15)	187,05(999)
1000;1000	0,3(15)	0,15(15)	8,95(46)	1000;100000	0,6(15)	0,3(15)	541,95(2262)
2000;1000	0,3(15)	0,45(15)	18,71(78)	2000;100000	0,15(15)	0,6(15)	1278,43(4681)
5000;1000	0,45(15)	0(0)	57,75(142)	5000;100000	0,75(15)	0,9(15)	3076,52(15709)
10000;1000	0,3(15)	0,45(15)	96,55(296)	10000;100000	1,5(15)	3,15(15)	-

Tabulka 6.10: Naměřené časy pro algoritmu *combo*

combo(konkurenční implementace)							
n;R	U.	W. C.	S. C.	n;R	U.	W. C.	S. C.
50;1000	0(0)	0(0)	0,15(15)	50;100000	0,15(15)	0,3(15)	10,19(125)
100;1000	0,15(15)	0,45(15)	0,6(15)	100;100000	0,15(15)	0(0)	6,54(78)
200;1000	0(0)	0(0)	0,9(15)	200;100000	0(0)	0(0)	10,02(125)
500;1000	0,3(15)	0,45(15)	1,35(15)	500;100000	0,15(15)	0,3(15)	7,52(31)
1000;1000	0(0)	0(0)	0,75(15)	1000;100000	0,15(15)	0,9(15)	7,53(78)
2000;1000	0,75(15)	0,6(15)	1,65(15)	2000;100000	0,15(15)	0,6(15)	7,35(15)
5000;1000	0,45(15)	0,75(15)	1,95(15)	5000;100000	0,45(15)	3,15(15)	7,81(31)
10000;1000	0,45(15)	0,3(15)	2,85(15)	10000;100000	1,35(15)	4,5(15)	7,35(15)

Tabulka 6.11: Naměřené časy pro algoritmus *DP1*

DP1 (konkurenční implementace)							
n;R	U.	W. C.	S. C.	n;R	U.	W. C.	S. C.
50;1000	1,11(3)	1,09(3)	1,17(3)	50;100000	189,39(391)	189,44(393)	188,59(433)
100;1000	6,41(14)	6,51(14)	6,51(14)	100;100000	749,63(1539)	749,81(1543)	754,15(1552)
200;1000	29,41(65)	29,51(63)	29,3(63)	200;100000	-	-	-
500;1000	188,6(374)	188,05(375)	190,53(378)	500;100000	-	-	-
1000;1000	757,76(1491)	754,76(1482)	750,23(1464)	1000;100000	-	-	-
2000;1000	3352,34(10862)	3351,22(12095)	3315,56(11865)	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

Tabulka 6.12: Naměřené časy pro algoritmus *DP2*

DP2 (konkurenční implementace)							
n;R	U.	W. C.	S. C.	n;R	U.	W. C.	S. C.
50;1000	0,33(1)	0,35(1)	0,43(1)	50;100000	70,68(151)	69,71(153)	74,89(183)
100;1000	2,48(6)	2,5(6)	2,93(6)	100;100000	284,88(609)	278,63(596)	319,25(671)
200;1000	11,19(25)	11,35(24)	11,83(25)	200;100000	1133,26(2477)	1112,45(2447)	1251,59(2531)
500;1000	71,01(145)	71,52(146)	71,41(144)	500;100000	-	-	-
1000;1000	285,41(579)	284,66(576)	278,72(561)	1000;100000	-	-	-
2000;1000	1139,11(2307)	1127,49(2286)	1103,61(2221)	2000;100000	-	-	-
5000;1000	-	-	-	5000;100000	-	-	-
10000;1000	-	-	-	10000;100000	-	-	-

6.4 Zhodnocení výsledků měření

V této sekci autor zhodnotí provedené měření.

6.4.1 Zhodnocení výsledků implementace algoritmu *MT1*

Z výsledků měření se ukazuje, že algoritmus *MT1* si počíná velice efektivně na nekorelovaných a slabě korelovaných datech, neboť nalezne dobře využitelné řešeni pro *prořezávání* a stavový prostor je tedy značně redukován. Na silně korelovaných datech už při malém počtu prvků umisťovaných do batohu existují instance, kdy v důsledku nenalezení dobře využitelného řešení k *prořezávání* algoritmus běží značně dlouhou dobu.

V paralelní verzi dochází většinou k zrychlování výpočtu. Toto zrychlení se liší napříč testovanými datovými sadami.

Z měření autor vypozoroval následující slabiny paralelní implementace. V některých případech, pokud jedna výpočetní jednotka má výpočetně náročnou úlohu a ostatní výpočetní jednotky od ní dostávají neustále výpočetně nenáročné úlohy, výkon algoritmu v důsledku neustálého přerozdělování zátěže časově degeneruje oproti běhu s jednou výpočetní jednotkou. Toto je možné pozorovat v tabulce 6.2 u slabě korelovaných dat s koeficientem R o hodnotě 100000 pro dvě vlákna. Navíc v důsledku ne optimálně navrženého sdílení zátěže, kdy výpočetní jednotky nežádají přímo jinou výpočetní jednotku o přidělení práce, zůstávají výpočetní jednotky určitý čas bez práce. Dále se může stát, že výpočetní jednotky hledají v částech stavového prostoru, kde se nenalézají lepší řešení a výpočetní čas je podobný běhu s jednou výpočetní jednotkou.

6.4.2 Zhodnocení výsledků implementace algoritmu *HS1974*

Z výsledku měření se ukazuje vliv míry korelace a vzrůstajících koeficientů atributů prvků, neboť se hůře uplatňuje vyřazování stavů na základě dominance stavů. Byla měřena verze s ukládáním konfigurace řešení a bez tohoto ukládání. Z měření je patrné, že ukládání konfigurace řešení, ačkoliv je paměťově šetrné, tvoří významnou část výpočetního času.

Ačkoliv pro malé instance paralelní režie při využití více vláken převažuje běh při vláknu jednom, pro dostatečně velké instance problému, je možné pozorovat následující zrychlení a efektivita (jako základ nebyla použita sekvenční implementace, neboť podává horší výpočetní časy nežli paralelní verze při jednom vláknu):

- Při použití dvou výpočetních jednotek dochází k zrychlení okolo hodnoty 1,7 a tedy efektivita na jednu výpočetní jednotku je okolo 85 %.
- Při použiti čtyř výpočetních jednotek dochází k zrychlení okolo hodnoty 2,6 a tedy efektivitě na jednu výpočetní jednotku okolo 65 %.

Pro efektivnější běh s více vlákny by bylo nutné pravděpodobně upravit algoritmus technikami omezujícími falešné sdílení, které autor neimplementoval.

Dalšího zrychlení by bylo možné dosáhnout paralelním zpracováním dvou nezávislých instancí při využití dekompozice instance. Autor od této paralelizace ustoupil z důvodů uvedených v části textu o implementaci. Nicméně s jinou technologií by tato možnost mohla být výhodná díky menšímu falešnému sdílení mezi výpočetními jednotkami, neboť dané dvě nezávislé instance mají oddělené datové struktury.

6.4.3 Zhodnocení výsledků implementace algoritmu *DPwL*

Vzhledem k tomu, že implementace *DPwL* byla využita v implementaci algoritmu *HS1974*, jsou zrychlení a efektivita při paralelizaci podobné. Při srovnání algoritmů *DPwL* s *HS1974* bez konfigurace je vidět přínos dekompozice problému na dvě nezávislé části.

Při použití v pokročilých algoritmech je *DPwL* kombinováno s mezemi jak bylo popsáno v sekci 3.2.4, což přináší redukci stavů v seznamech a tedy urychlení výpočtu, autor bohužel z časových důvodů tuto verzi neimplementoval a v práci tedy toto srovnání není.

6.4.4 Zhodnocení výsledků implementace algoritmu algoritmu hrubé síly

Tento přístup se ukázal v sekvenční i paralelní verzi jako neefektivní již při nejmenších testovaných instancích o 50 prvcích. Autor proto výsledky pro tento přístup neuvádí.

6.4.5 Zhodnocení výsledků konkurenční implementace algoritmu *minknap*

Z výsledků měření vychází tento algoritmus jako velice efektivní na většině testovaných datových sadách. K menšímu nárůstu výpočetního času dochází pro instance silně korelovaných dat s R o hodnotě 1000, pro R o hodnotě 100000 je již nárůst relativně vysoký.

6.4.6 Zhodnocení výsledků konkurenční implementace algoritmu *combo*

Tento algoritmus zvládl při malém výpočetním čase všechny autorem testované datové sady. Tento algoritmus ukázal jako nejefektivnější z testovaných algoritmů. Nicméně je třeba dodat, že dle [3] existují typy instancí, kde si tento algoritmus počíná hůře.

6.4.7 Zhodnocení výsledků konkurenční implementace algoritmu *DP1*

Tento algoritmus poskytuje konfiguraci výsledného řešení a tedy nebude porovnán s autorovými implementacemi neposkytujícími konfiguraci optimálního řešení. Oproti algoritmu *MT1* si tento algoritmus počíná lépe pouze na silně korelovaných datech. Oproti algoritmu *HS1974* s konfigurací si *DP1* počíná lépe na všech datových sadách kromě nekorelovaný a slabě korelovaných s R o hodnotě 100000.

6.4.8 Zhodnocení výsledků konkurenční implementace algoritmu *DP2*

Protože tento algoritmus neposkytuje konfiguraci optimálního řešení nebude porovnán s autorovými implementacemi dynamického programování poskytujícími tuto konfiguraci. Pro algoritmus platí v porovnání s algoritmem *MT1* to samé jako pro *DP1*, tedy že si počíná lépe na silně korelovaných datech. Při porovnání s algoritmy *DPwL* a *HS1974* bez konfigurace platí, že si počíná lépe na všech datových sadách kromě nekorelovaný a slabě korelovaných s R o hodnotě 100000.

6.4.9 Shrnutí

Z autorových implementací je pro nekorelovaná a slabě korelovaná data díky *prořezávání* stavového prostoru nejefektivnější implementace algoritmu *MT1*, oproti tomu implementace algoritmu *HS1974* si touto technikou nemůže pomoci a její výpočetní časy jsou v důsledku toho výrazně vyšší.

Pro silně korelovaná data implementace algoritmu *HS1974* podává lepší výsledky nežli implementace algoritmu *MT1*, neboť u ní nedochází k velkému

nárůstu výpočetního času jako pro některé instance u algoritmu *MT1* v důsledku malé efektivity *prořezávání*.

Algoritmy *DP1* a *DP2* jsou oproti autorovým implementacím efektivnější na určitých datových sadách.

Konkurenční implementace algoritmu *minknap* a *combo* se ukázaly výrazně efektivnější na všech testovaných datových sadách než autorem implementované algoritmy i při využití paralelizace.

Autorem implementované algoritmy se tedy ve srovnání s konkurencí ukázali jako méně výkonné, nicméně algoritmy *minknap* a *combo* využívají jako svoji součást algoritmus *DPwL* a je otázkou jestli by při začlenění autorovy paralelizace nebyl jejich běh zrychlen. Dále součástí algoritmu *MTHard*, pro který není uvedeno měření, neboť autor nenalezl jeho implementaci, je algoritmus *MT1*, a při využití autorovy paralelizace by mohlo dojít k jeho zrychlení.

Závěr

Cílem teoretické části práce bylo popsat problém batohu 0-1, přístupy k jeho exaktnímu řešení a vybrané algoritmy. Cílem praktické části práce byla implementace a paralelizace vybraných algoritmů. Dále porovnání výpočetních časů autorových implementací a konkurenčních algoritmů na vybraných datových sadách.

V práci byl popsán problém batohu 0-1 a dále byly stručně popsány přístupy používané k hledání exaktního řešení problému batohu 0-1. Následně jsou popsány vybrané algoritmy pro které je navrhnuta paralelizace. Tyto algoritmy jsou implementovány v sekvenční verzi a poté v paralelní verzi postavené na navržených paralelizacích. Na závěr jsou změřeny výpočetní časy autorových a konkurenčních implementací na vybraných datových sadách a výsledky porovnány. Na práci by bylo možné navázat začleněním paralelizací do pokročilých algoritmů.

Bibliografie

1. CHINNECK, John W. Practical optimization: a gentle introduction. *Systems and Computer Engineering), Carleton University, Ottawa. http://www. sce. carleton. ca/faculty/chinneck/po. html*. 2006.
2. MARTELLO, Silvano; TOTH, Paolo. *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN 0-471-92420-2.
3. PISINGER, David. Where are the hard knapsack problems? *Computers and Operations Research*. 2005, roč. 32, č. 9, s. 2271–2284. ISSN 0305-0548. Dostupné z DOI: `https://doi.org/10.1016/j.cor.2004.03.002`.
4. MARTELLO, Silvano; PISINGER, David; TOTH, Paolo. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*. 2000, roč. 123, s. 325–332.
5. MARTELLO, Silvano; TOTH, Paolo. Upper Bounds and Algorithms for Hard 0-1 Knapsack Problems. *Operations Research*. 1997, roč. 45, s. 768–778. Dostupné z DOI: `10.1287/opre.45.5.768`.
6. PISINGER, David. A Minimal Algorithm for the 0-1 Knapsack Problem. *Operations Research*. 2002, roč. 45. Dostupné z DOI: `10.1287/opre.45.5.758`.
7. MARTELLO, Silvano; PISINGER, David; TOTH, Paolo. Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem. *Management Science*. 1999, roč. 45, s. 414–424. Dostupné z DOI: `10.1287/mnsc.45.3.414`.
8. MARTELLO, S.; TOTH, P. *Knapsack problems: algorithms and computer implementations*. J. Wiley & Sons, 1990. Wiley-Interscience series in discrete mathematics and optimization. ISBN 9780471924203. Dostupné také z: `https://books.google.cz/books?id=0dhQAAAAMAAJ`.

9. INGARGIOLA, Giorgio P.; KORSH, James F. Reduction Algorithm for Zero-One Single Knapsack Problems. *Management Science*. 1973, roč. 20, č. 4, s. 460–463. ISSN 00251909, 15265501. ISSN 00251909, 15265501. Dostupné také z: http://www.jstor.org/stable/2629626.
10. MORRISON, David R.; JACOBSON, Sheldon H.; SAUPPE, Jason J.; SEWELL, Edward C. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*. 2016, roč. 19, s. 79–102. ISSN 1572-5286. Dostupné z DOI: https://doi.org/10.1016/j.disopt.2016.01.005.
11. CLAUSEN, Jens. Branch and Bound Algorithms - Principles and Examples. *Computer*. 1999.
12. KOLESAR, Peter. A Branch and Bound Algorithm for the Knapsack Problem. *Management Science*. 1967, roč. 13, s. 723–735. Dostupné z DOI: 10.1287/mnsc.13.9.723.
13. LEVITIN, Anany V. *Introduction to the Design and Analysis of Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201743957.
14. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN 0262033844, 9780262033848.
15. KELLERER, H.; PISINGER, H.K.U.P.D.; PFERSCHY, U.; PISINGER, D. *Knapsack Problems*. Springer, 2004. Springer Nature Book Archives Millennium. ISBN 9783540402862. Dostupné také z: https://books.google.cz/books?id=u5DB7gck08YC.
16. BELLMAN, Richard. The theory of dynamic programming. *Bull. Amer. Math. Soc.* 1954, roč. 60, č. 6, s. 503–515. Dostupné také z: https://projecteuclid.org:443/euclid.bams/1183519147.
17. BALAS, Egon; ZEMEL, Eitan. An Algorithm for Large Zero-One Knapsack Problems. *Operations Research*. 1980, roč. 28, s. 1130–1154. Dostupné z DOI: 10.1287/opre.28.5.1130.
18. MARTELLO, Silvano; TOTH, Paolo. A New Algorithm for the 0-1 Knapsack Problem. *Management Science*. 1988, roč. 34, s. 633–644. Dostupné z DOI: 10.1287/mnsc.34.5.633.
19. HOROWITZ, Ellis; SAHNI, Sartaj. Sahni, S.: Computing partitions with applications to the knapsack problem. Journal of the ACM 21, 277-292. *J. ACM*. 1974, roč. 21, s. 277–292. Dostupné z DOI: 10.1145/321812.321823.
20. MARTELLO, Silvano; TOTH, Paolo. An Upperbound for the Zero-One Knapsack Problem and a Branch and Bound Algorithm. *European Journal of Operational Research*. 1977, roč. 1, s. 169–175. Dostupné z DOI: 10.1016/0377-2217(77)90024-8.

21. MCCOOL, Michael; REINDERS, James; ROBISON, Arch. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN 9780123914439, 9780124159938.

22. *Parallel Algorithm - Models [online]*. 2019. Dostupné také z: `https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_models.htm`. [cit. 2019-04-21].

23. TVRDÍK, Pavel. *Úvod do paralelního a distribuovaného programování [online]*. 2019. Dostupné také z: `https://courses.fit.cvut.cz/MI-PDP/media/lectures/MI-PDP-Prednaska01-Introduction.pdf`. [cit. 2019-04-21].

24. GRAMA, Ananth; KARYPIS, George; KUMAR, Vipin; GUPTA, Anshul. *Introduction to Parallel Computing (2nd Edition)*. 2003. ISBN 0201648652.

25. KINDERVATER, G.A.P.; LENSTRA, J.K. An introduction to parallelism in combinatorial optimization. *Discrete Applied Mathematics*. 1986, roč. 14, č. 2, s. 135–156. ISSN 0166-218X. Dostupné z DOI: `https://doi.org/10.1016/0166-218X(86)90057-0`.

26. GHOSH, Sumit. *Print all subsequences of a string [online]*. 2019. Dostupné také z: `https://www.geeksforgeeks.org/print-subsequences-string/`. [cit. 2019-04-21].

27. PISINGER, David. *David Pisinger's optimization codes [online]*. 2019. Dostupné také z: `http://hjemmesider.diku.dk/~pisinger/codes.html`. [cit. 2019-04-28].

28. TRIVEDI, Utkarsh. *Implementation of 0/1 Knapsack using Branch and Bound [online]*. 2019. Dostupné také z: `https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/`. [cit. 2019-04-28].

29. PAL, Soumik. *Printing Items in 0/1 Knapsack [online]*. 2019. Dostupné také z: `https://www.geeksforgeeks.org/printing-items-01-knapsack/`. [cit. 2019-04-28].

30. MISHRA, Shashank. *A Space Optimized DP solution for 0-1 Knapsack Problem [online]*. 2019. Dostupné také z: `https://www.geeksforgeeks.org/space-optimized-dp-solution-0-1-knapsack-problem/`. [cit. 2019-04-28].

Příloha **A**

Seznam použitých zkratek

DFS prohledávací algoritmus *depth first search*

DPwL algoritmus *Dynamic programming with lists*

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `datasets` ..... datové sady použité pro testování
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF