Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Nina Vehovec

Hibridizacija požrešnih algoritmov in hitrega urejanja

diplomsko delo

interdisciplinarni univerzitetni
študijski program
prve stopnje
računalništvo in matematika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2017

To delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva-Deljenje pod enakimi pogoji 2.5 Slovenija* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani creativecommons.si ali na Inštitutu za intelektualno lastnino, Streliška 1, 1000 Ljubljana.

Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco GNU General Public License, različica 3 (ali novejša). To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani `http://www.gnu.org/licenses/`.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Eden izmed pomembnejših pristopov k reševanju računskih problemov so požrešni algoritmi, ki pogosto kot predprocesiranje uporabljajo urejanje podatkov. Pri tem se največkrat uporablja hitro urejanje, ker je izredno učinkovito. Poleg tega pa omogoča, da nekatere požrešne algoritme lahko direktno vgradimo vanj tako, da se urejanje ne izvede do konca, ampak le toliko, kot je potrebno, da poiščemo rešitev problema. Tako predelani algoritem lahko v nekaterih primerih deluje precej hitreje. V okviru diplomske naloge je potrebno poiskati požrešne algoritme, ki so primerni za ta način predelave in jih predelati. Nato pa predelave na primernem naboru testnih primerov eksperimentalno ovrednotiti.

Zahvaljujem se mentorju doc. dr Juriju Miheliču za potrpežljivost, strokovnost, vse komentarje, nasvete in njegov dragocen čas. Zahvala velja tudi ostalim profesorjem, saj brez znanj, ki sem jih pridobila v teku šolanja, ne bi uspela narediti diplomske naloge. Hvala tudi vsem sorodnikom za podporo.

Kazalo

Povzetek

Naslov: Hibridizacija požrešnih algoritmov in hitrega urejanja

Avtor: Nina Vehovec

Povzetek: Požrešna metoda je ena izmed najbolj uporabljenih tehnik pri načrtovanju algoritmov, za katero obstaja več vrst optimizacij. Naša ideja je bila optimiziranje požrešnih algoritmov s pomočjo algoritma hitrega urejanja. V algoritem hitrega urejanja smo vstavili požrešni algoritem. Na tak način ustvarjen hibridni algoritem, lahko na določenih primerih deluje precej hitreje kot navadni požrešni algoritem. Hibridizacijo smo preizkusili na problemih preprostega nahrbtnika, menjave kovancev in izbora aktivnosti ter na Kruskalovem algoritmu in pri razvrščanju poslov v delavnici z enim strojem. Hibridni algoritem je deloval bolje od navadnega požrešnega algoritma pri preprostem problemu nahrbtnika in pri problemu menjave kovancev.

Ključne besede: požrešno, nahrbtnik, posli, aktivnosti, Kruskal, kovanci, urejanje, hibridizacija, algoritmi.

Abstract

Title: Hybridization of greedy algorithms and quicksort

Author: Nina Vehovec

Abstract: The greedy method is one of the most commonly used techniques in algorithm design and there are many different optimizations available. This thesis presents one of them. Our idea for optimization was improving the running time of greedy algorithms with the help of the quicksort algorithm. We integrated the greedy algorithm into quicksort to produce a hybrid algorithm, which can solve certain problems significantly faster than the normal greedy algorithm. In the thesis we chose to test hybridization on the activity-selection problem, the fractional knapsack problem, the coin changing problem, Kruskal's algorithm and unit-task scheduling. We experimentally confirmed that hybrid algorithms (do) indeed perform better with the coin changing problem and the fractional knapsack problem.

Keywords: greedy, hybridization, algorithms, activity-selection, knapsack, Kruskal, coin-changing, quicksort, unit-tasks.

Poglavje 1

Uvod

Požrešna metoda je poleg dinamičnega programiranja, pretokov in metode deli in vladaj, ena izmed najbolj uporabljenih metod pri načrtovanju algoritmov. Z njo rešujemo predvsem optimizacijske probleme. Uporabna je tudi, ker pokriva širok spekter področij. Poznamo razne požrešne algoritme na grafih, huffmanove kode, požrešni algoritmi rešujejo tudi probleme razvrščanja elementov in gručenja in še bi lahko naštevali.

Tako kot računalništvo se tudi algoritmi konstantno razvijajo in spreminjajo. V praksi se vedno išče hitrejše izvajanje algoritmov, torej njihova optimizacija. V diplomi smo se lotili podobnega problema, saj je bil naš cilj izboljšati delovanje izbranih požrešnih algoritmov na določenem naboru podatkov. Hitrost hibridiziranega požrešnega algoritma smo primerjali z navadnim požrešnim algoritmom in rezultate prikazali z grafom. Izbrani požrešni algoritmi so morali vsebovati urejanje elementov, ker smo hibridizirali požrešne algoritme z algoritmom hitrega urejanja. Taka hibridizacija lahko v določenih primerih deluje precej hitreje.

V diplomski nalogi najprej predstavimo idejo požrešnih algoritmov, nato opišemo hitro urejanje in učinkovito implementacijo disjunktnih množic. V poglavju 2 tudi predstavimo generični algoritem za hibridizacijo algoritma hitrega urejanja in poljubnega požrešnega algoritma, ki kot predprocesiranje uporablja urejanje elementov. Nato prikažemo tudi malenkostno modificiran

generični algoritem s primerom izvajanja.

V 3. poglavju naredimo pregled izbranih požrešnih algoritmov, kjer pri vsakem najprej opišemo problem, ki ga rešujemo. Nato predstavimo ustrezni požrešni algoritem, sledi hibridizacija in na koncu oba algoritma eksperimentalno ovrednotimo. Po pregledu vseh algoritmov sledi še zaključek s sklepnimi ugotovitvami.

Poglavje 2

Teoretično ozadje

2.1 Požrešni algoritmi

V knjigi [1] navajajo šah kot primer igre, kjer je načrtovanje potez ključno za dosego zmage. Poznamo tudi veliko drugih iger, ena od njih je Scrabble, kjer je za solidno igro dovolj sprejemanje najboljše odločitve v danem trenutku. Slednjemu principu sledi tudi požrešna metoda.

Ideja požrešnega algoritma je, da rešitev gradi postopoma [2]. Pri tem na vsakem koraku v rešitev doda tisti element, ki trenutno pripomore k največjemu dobičku. Takšen pristop sicer ni vedno najboljši, vendar obstaja mnogo zanimivih problemov, kjer na takšen način lahko pridobimo optimalno rešitev.

Optimalnost požrešnega algoritma [3] določata optimalna podstruktura (angl. *optimal substructure*) in požrešna izbira (angl. *greedy-choice property*). Požrešna izbira nam zagotavlja, da več lokalnih odločitev, kjer vedno izberemo trenutno najboljšo rešitev, vodi do globalne optimalne rešitve. Torej, glede na to, da vedno izberemo odločitev, ki se nam trenutno najbolj izplača, nato rešujemo podprobleme, ki ob tem nastanejo, je trenutna požrešna odločitev lahko odvisna samo od preteklih odločitev in ne od rešitev podproblemov. Pri tem moramo biti pozorni, da vsaka izbrana odločitev, vodi do globalne optimalne rešitve. Če ima problem optimalno rešitev, ki vsebuje

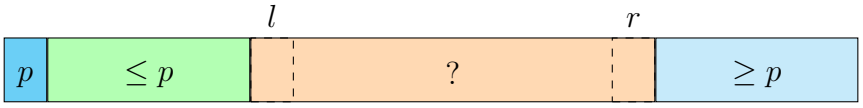
optimalne rešitve podproblemov, pravimo, da ima optimalno podstrukturo.

V diplomi se bomo osredotočili na požrešne algoritme, ki so optimalni in kot predprocesiranje uporabljajo urejanje podatkov, saj je za našo hibridizacijo ključna uporaba hitrega urejanja.

2.2 Hitro urejanje

Hitro urejanje se v praksi zelo pogosto uporablja, ker je razmeroma učinkovito. V povprečju je časovna zahtevnost hitrega urejanja $O(n \log n)$. Obstaja več različic algoritma, mi bomo predstavili samo za eno, saj za potrebe diplomske naloge to zadostuje.

Delovanje hitrega urejanja, kjer uporabljamo križanje kazalcev [4] bomo prikazali s pomočjo slik. Elemente urejamo v nepadajočem vrstnem redu. Tekom izvajaja algoritma bomo prišli do stanja, ki je prikazano na sliki 2.1.



Slika 2.1: Vmesno stanje

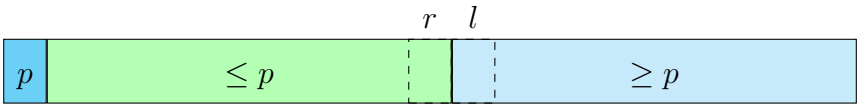
V tabeli smo določili pivot p, z zeleno je označen del, kjer so elementi manjši ali enaki pivotu p, z oranžno je predstavljen še nepregledan del elementov in z modro smo označili elemente, ki so večji ali enaki pivotu. Indeksa oz. kazalca l in r kažeta na najbolj levi in desni del neobdelane tabele.

Pri vsakem porazdeljevanju l pomikamo desno, dokler je element, na katerega kaže l manjši ali enaki pivotu. Podobno indeks r pomikamo levo, dokler je element na katerega kaže r večji ali enak indeksu. Ko l kaže na element večji od pivota in r na element manjši od pivota, elementa zamenjamo.

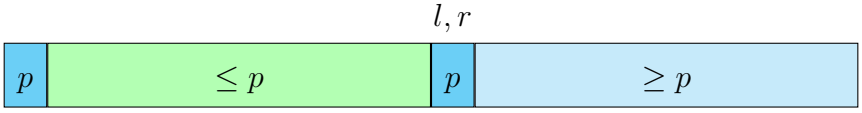
To ponavljamo dokler se indeksa (kazalca) ne križata, torej takrat velja $r \leq l$. Pri tem dobimo dve možni končni stanji.

Na sliki 2.2 je prikazano stanje, kjer sta se kazalca dejansko prekrižala, torej velja $r < l$. Na sliki 2.3 je prikazano stanje, ko sta se kazalca samo

staknila in kjer velja $l = r$.

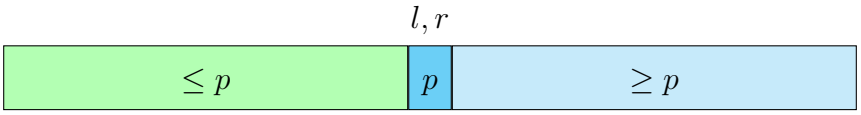


Slika 2.2: Stanje, kjer sta se indeksa l in r prekrižala



Slika 2.3: Stanje, kjer kazalca l in r sovpadata

Ker za indeks l velja, da kaže na element večji ali enak pivotu in za r, da kaže na element manjši ali enak pivotu, sklepamo, da je element na mestu stika enak pivotu. V obeh primerih nato še zamenjamo pivot p in element na mestu r. S tem je porazdeljevanje končano. Dobili smo tabelo, ki ima na levi strani elemente manjše ali enake pivotu, elementi na desni strani so večji ali enaki pivotu. Končno stanje porazdeljevanja lahko vidimo na sliki 2.4.



Slika 2.4: Stanje po končanem porazdeljevanju

Hitro urejanje nato rekurzivno uredi še levi (zeleni) in desni (svetlo modri) del tabele.

2.3 Disjunktne množice

Včasih se v računalništvu pojavi potreba po implementaciji paroma disjunktnih množic. Eden od najpogostejših primerov uporabe te strukture je pri algoritmih, ki nek problem razdelijo dinamično na podprobleme. Te probleme nato rešijo in na koncu združijo rešitve. Drug primer uporabe [2] je pri

grafih, kjer lahko postopoma gradimo graf z dodajanjem povezav. Ta način uporabljamo ravno pri Kruskalovem algoritmu.

Recimo, da grupiramo n različnih elementov v disjunktne množice. Pogosto bomo nato želeli ugotoviti del katere množice je naš element ali pa bomo želeli množici preprosto združiti.

Cilj te implementacije bo tako ravno omogočiti hitro iskanje in posodabljanje množic.

Naj bo $\mathscr{S} = \{S_1, S_2, ..., S_k\}$ družina paroma disjunktnih množic [3]. Za vsako množico S_i definiramo predstavnika, ki je nek element te množice. Predstavnik je lahko poljuben element ali pa točno določen, odvisno od problema, ki ga rešujemo. Definirajmo tudi možne operacije na množicah:

MAKE-SET(x) ustvari novo množico, katere edini element in torej tudi predstavnik je x. Ob tem mora veljati, da x ni del katere druge množice. Če imamo n različnih elementov in za vsakega ustvarimo svojo množico je skupna časovna zahtevnost vseh teh operacij $O(n)$.

FIND-SET(x) vrne predstavnika množice, ki vsebuje element x. Časovna zahtevnost te operacije je odvisna od njene implementacije. Lahko je $O(\log n)$ ali pa celo $O(1)$.

UNION(x, y) združi množici, ki vsebujeta elementa x in y. Pri tem predpostavimo, da sta množici z elementoma x in y disjunktni. Naj bo p_x predstavnik množice z elementom x in p_y predstavnik množice z elementom y. Za predstavnika si lahko izberemo katerikoli element nove množice, čeprav v praksi ponavadi izberemo kar p_x ali p_y. Časovna zahtevnost operacije UNION je lahko $O(n)$ ali tudi $O(n \log n)$, odvisno od implementacije.

2.3.1 Disjunktne množice gozdov

Disjunktne množice lahko implementiramo na več načinov. Ena od možnih implementacij so povezani seznami, obstaja pa tudi druga, in sicer hitrejša implementacija, kjer predstavimo množice z drevesi [3].

Pri implementaciji disjunktnih množic z gozdom vsako vozlišče v drevesu vsebuje en element in vsako drevo predstavlja eno množico v gozdu. Ko-

ren drevesa je predstavnik in tudi sam sebi starš, vozlišča v drevesu imajo povezave samo do svojih staršev. Funkcija 2.3.1: MAKE-SET ustvari drevo z enim samim vozliščem, funkcija 2.3.2: FIND-SET nam vrne koren, torej predstavnika drevesa, in funkcija 2.3.3: UNION poveže drevesi tako, da eno drevo kaže na koren drugega.

Vendar to ni dovolj za izboljšanje časovne zahtevnosti implementacije. Ključna je uporaba izboljšav, kjer drevesa združujemo glede na njihov rang (angl. *union by rank*) in kompresija poti (angl. *path compression*).

Pri združevanju dreves glede na njihov rang, uporabimo poenostavljen pristop, kjer namesto, da bi za vsako vozlišče x hranili velikost poddrevesa, hranimo kar samo njegov rang, t.j. dolžina preproste poti od vozlišča x do najbolj oddaljenega lista poddrevesa, katerega koren je x, torej bi lahko rekli, daje rang kar višina vozlišča.

V funkciji 2.3.3: UNION nato koren drevesa z manjšim rangom povežemo na koren drevesa, ki ima večji rang. Če sta ranga dreves enaka izberemo eno drevo za starša in povečamo njegov rang.

Izboljšavo kompresije poti pa uporabimo v funkciji 2.3.2: FIND-SET, kjer vsako vozlišče, ki ga obiščemo na poti do korena, povežemo tako, da kaže kar direktno na koren drevesa (predstavnik drevesa), pri tem rangov vozlišč ne spreminjamo.

Function 2.3.1

```
1: MAKE-SET(x)
2:     x.p ← x
3:     x.rank ← 0
```

Function 2.3.2

```
1: FIND-SET(x)
2:     if x ≠ x.p then x.p ← FIND-SET(x.p)
3:     return x.p
```

Sedaj, ko smo definirali izboljšavi, lahko prikažemo delovanje funkcij MAKE-SET, FIND-SET in UNION z uporabljenimi izboljšavami. Kot smo

že omenili, funkcija 2.3.1: MAKE-SET ustvari drevo z enim vozliščem, ob tem inicializira njegov rang na 0 in sebe določi za starša. Funkcija 2.3.2: FIND-SET vedno vrne starša elementa x. Če starš ni kar element sam, nam rekurziven klic s parametrom $x.p$ vrne koren drevesa. V vrstici 2 torej elementu x spremenimo starša na koren. V funkciji 2.3.3: UNION pokličemo funkcijo 2.3.4: LINK, ki prejme korena dveh dreves, ki ju nato združimo glede na rang.

Function 2.3.3

```
1: UNION(x, y)
2:     LINK(FIND-SET(x), FIND-SET(y))
```

Function 2.3.4

```
1: LINK(x, y)
2:     if x.rank > y.rank then y.p ← x
3:     else
4:         x.p ← y
5:         if x.rank = y.rank then y.rank ← y.rank + 1
```

2.4 Hibridizacija požrešnega algoritma

Algoritmi, ki smo jih v diplomski nalogi implementirali, so morali kot predprocesiranje uporabljati urejanje podatkov, da smo hibridizacijo sploh lahko izvedli. Pri tem smo sledili generičnemu algoritmu 2.4.2, ki vzame algoritem hitrega urejanja in ga modificira tako, da vanj vgradi požrešni algoritem.

Hibridna funkcija 2.4.2 izvaja urejanje elementov, dokler ne doseže robnih pogojev, v katerih se izvede požrešni del hibridnega algoritma. Tudi funkcija 2.4.1: CONSIDER predstavlja požrešni del algoritma.

Odločili smo se, da bomo v generičnem algoritmu prikazali urejanje elementov v nepadajočem vrstnem redu. Zaradi lažje in bolj jasne predstavitve algoritma je spremenljivka *greedy_variables* globalna. Spremenljivka

greedy_variables v dejanski implementaciji lahko predstavlja tabelo, eno spremenljivko ali pa tudi več različnih spremenljivk, odvisno od zahtev algoritma, ki ga implementiramo. Ponavadi nam ta spremenljivka predstavlja neko vrednost, ki jo želimo doseči tekom izvajanja algoritma, oziroma neko vrednost, lahko tudi vrednosti, na podlagi katere se odločimo ali bomo trenuten element dodali v rešitev ali ne. Tekom izvajanja algoritma se tako spremenljivka velikokrat spreminja

Function 2.4.1

```
1: CONSIDER(result, element)
2:     if GREEDY-CONDITION(element, greedy_variables) then
3:         result ← result ∪ {item}
4:         update greedy_variables
```

Največji del požrešnega algoritma se izvaja ravno v pomožni funkciji 2.4.1: CONSIDER, ki jo pokliče rekurzivni algoritem. Parametra funkcije sta kazalec na rešitev *result* in element *element*, ki ga želimo dodati v rešitev. Da lahko dodamo element v rešitev, moramo najprej preveriti, če ustreza pogojem, ki jih določa izbrani požrešni algoritem (vrstica 2). Če so pogoji izpolnjeni, vrstici 3-4 dodata element v rešitev in ustrezno posodobita spremenljivko *greedy_variables*.

Hibridna funkcija 2.4.2 kot parametre prejme tabelo elementov *elements* in indeksa $left$ in *right*, v območju katerih v tej iteraciji rekurzije urejamo tabelo *elements*.

Kot smo že omenili, hibridna funkcija 2.4.2 vsebuje urejanje, torej rekurzivni del hibridiziranega algoritma. Ta algoritem ima več robnih pogojev in prvega lahko vidimo že v vrstici 2, kjer kot rešitev samo vrnemo prazno množico, saj smo že pregledali vse elemente med indeksoma $left$ in *right*. V vrstici 3 določimo pivotni element, ki je najbolj levi element trenutne tabele. Če je pogoj v vrstici 4 izpolnjen, potem vrstice 5-7 inicializirajo spremenljivko *result*, pokličejo funkcijo 2.4.1: CONSIDER in vrnejo rezultat *result*. Funkcija 2.4.1: CONSIDER poskuša dodati v rešitev element $elements[r+1]$. Vrstice 8-16 nato urejajo elemente *elements*.

Function 2.4.2

```
1:  HYBRIDIZED-GREEDY-ALGORITHM(elements, left, right)
2:      if right < left then return ∅
3:      pivot ← elements[left]
4:      if right == left then
5:          result ← ∅
6:          CONSIDER(result, pivot)
7:          return result
8:      l ← left
9:      r ← right
10:     while l ≤ r do
11:         while elements[l].key < pivot.key do l ← l + 1
12:         while elements[r].key > pivot.key do r ← r − 1
13:         if l > r then break
14:         SWAP(i, l, r)
15:         l ← l + 1
16:         r ← r − 1
17:     leftsol ← HYBRIDIZED-GREEDY-ALGORITHM(elements, left, r)
18:     if l − r == 2 then CONSIDER(leftsol, elements[r + 1])
19:     if STOP-CONDITION then return leftsol
20:     rightsol ← HYBRIDIZED-GREEDY-ALGORITHM(elements, l, right)
21:     return leftsol ∪ rightsol
```

Omembe vredni sta vrstici 11 in 12, kjer lahko vidimo, da urejamo elemente v nepadajočem vrstnem redu, glede na ključ key elementa $elements[l]$ oziroma $elements[r]$. V vrstici 17 nato pokličemo levi del rekurzije, torej za parametra med drugim vzamemo indeksa $left$ in r. Drugi pogoj, ki mora biti izpolnjen, da lahko preidemo na požrešni del hibridnega algoritma lahko vidimo v vrstici 18. Tu zopet poskušamo v rešitev dodati pivotni element. Če smo našli optimalno rešitev, se v 19. izvede pogoj STOP-CONDITION, kjer nato samo vrnemo spremenljivko $leftsol$, kar pomeni, da nam ni potrebno izvajati še desnega dela rekurzije. V 20. vrstici izvedemo desni del rekurzije, nato v vrstici 21 vrnemo združeni rešitvi obeh rekurzivnih klicev.

Poglavje 3

Izbrani hibridizirani požrešni algoritmi

V tem poglavju bomo predstavili izbrane požrešne algoritme, nato pa še njihovo implementacijo ter rezultate eksperimentov, ki smo jih izvedli.

3.1 Testno okolje

Požrešne in hibridne algoritme smo napisali v programskem jeziku C. Nato smo izmerili njihovo hitrost izvajanja v milisekundah. Merili smo čas, ki ga je za izvajanje porabila centralna procesna enota. V programskem jeziku C lahko to preprosto izmerimo s pomočjo funkcije *clock()*. Pri tem smo uporabljali prevajalnik GCC brez optimizacije.

Testiranje smo izvedli na operacijskem sistemu Windows 7, na računalniku s procesorjem Intel(R) Core(TM) i5-3360M z dvemi fizičnimi jedri in 8 GB delovnega pomnilnika.

3.2 Problem menjave kovancev

3.2.1 Definicija problema

S problemom menjave kovancev se dnevno srečujejo prodajalci v trgovinah. Zaloge kovancev so omejene, torej morajo vsako menjavo denarja izvesti z najmanjšim možnim številom kovancev [5].

Naj bo spremenljivka *goal* ciljni znesek, ki ga morajo vrniti stranki in S množica možnih vrednosti kovancev. Predpostavimo uporabo evrskih kovancev [6], kjer mora za vsaki vrednosti x_i in x_j veljati, da je ena vsaj dvakrat večja od druge, torej $x_j \geq 2x_i$, če želimo problem rešiti optimalno. Recimo, da imamo množico vrednosti $\{1, 2, 5, 7, 10\}$ in ciljni znesek 14 centov, ter neomejeno število kovancev z različnimi vrednostmi. Požrešni algoritem brez zgornjega pogoja, bi za rešitev izbral en kovanec z desetimi centi in dva kovanca po dva centa, čeprav optimalna rešitev problema vsebuje dva kovanca s sedmimi centi.

Torej pri problemu menjave kovancev želimo z najmanjšim številom kovancev doseči določen znesek.

3.2.2 Požrešni algoritem

Problem lahko optimalno rešimo s požrešno metodo, če pred izvedbo požrešnega algoritma vrednosti kovancev uredimo v nenaraščajočem vrstnem redu.

Požrešni algoritem prejme množico kovancev *coins* in ciljni znesek *goal*, nato pa v vsaki iteraciji zanke izbere kovanec z največjo vrednostjo in ga doda v množico C [7]. Ciljnega zneska *goal* pri tem ne sme prekoračiti.

Delovanje požrešnega algoritma je prikazano v funkciji 3.2.1. V vrsticah 2-3 inicializiramo množico rešitev C na prazno množico in določimo vrednost spremenljivke n. Zanka *for* se v vrsticah 5-8 sprehodi čez urejene vrednosti kovancev *coins*. Če vrednost kovanca ne presega ciljnega zneska, kovanec dodamo v rešitev in zmanjšamo ciljni znesek *goal*.

Function 3.2.1

```
1: GREEDY-COIN-CHANGING(goal, coins)
2:     C ← ∅
3:     n ← length(coins)
4:     sort the coins coins into nonincreasing order by denomination
5:     for i ← 1 . . . n do
6:         if coins[i] ≤ goal then
7:             goal ← goal − coins[i]
8:             C ← C ∪ {coins[i]}
9:     return C
```

Požrešni algoritem za problem menjave kovancev ima časovno zahtevnost enako kot algoritem hitrega urejanja, torej $O(n \log n)$.

3.2.3 Hibridni požrešni algoritem

Hibridni algoritem za problem menjave kovancev natančno sledi generičnemu hibridnemu algoritmu, ki je bil predstavljen v razdelku 2.4.

Naj omenimo, da smo generično spremenljivko *greedy_variables* v tem hibridnem algoritmu zaradi večje razumljivosti algoritma nadomestili z *goal*, in sicer ta spremenljivka hrani vsoto, ki jo želimo doseči z najmanjšim možnim številom kovancev *coins*. Prav tako je globalna in se tekom izvajanja vsakokrat, ko dodamo kovanec oz. njegovo vrednost, zmanjša.

Generično ime spremenljivke *elements* smo zaradi razumljivosti algoritma zamenjali z imenom *coins*. Podobno smo zamenjali tudi spremenljivko *element* s *coin*. Spremenljivka *coins* vsebuje vrednosti kovancev, ki so potenciali za rešitev.

Tako kot generična spremenljivka *greedy_variables*, se tudi pogoj STOP-CONDITION zelo razlikuje od izbranega požrešnega algoritma. Ta je ključen tudi za hitrost hibridiziranega algoritma. Pri problemu menjave kovancev ga tako nadomestimo s preprostim pogojem, ki preveri ali je spremenljiva *goal* manjša ali enaka 0, kar pomeni, da smo že dosegli želeni znesek.

Function 3.2.2

```
1: CONSIDER-COIN(result, coin)
2:     if coin ≤ goal then
3:         result ← result ∪ {coin}
4:         goal ← goal − coin
```

Funkcija 3.2.2 je prilagoditev splošne funkcije 2.4.1: CONSIDER za problem menjave kovancev. Kot parameter prejme kazalec na rešitev *result* in kovanec oz. njegovo vrednost *coin*. Če je vrednost kovanca *coin* manjša ali enaka spremenljivki *goal*, potem v vrsticah 3-4 kovanec dodamo v rešitev in zmanjšamo vrednost spremenljivke *goal* za vrednsot *coin*, enako kot v požrešnem algoritmu.

Funkcija 3.2.3 je rekurzivna hibridna funkcija z vhodnimi parametri *coins* in indeksema $left$ in $right$. Ker je zelo podobna generični funkciji 2.4.2 bomo omenili samo nekaj manjših razlik. V 3. vrstici določimo pivot tako kot pri generičnem algoritmu. Vzamemo tabelo *coins* in element na indeksu $left$ je naš pivot. Vrstici 11 in 12 sta del urejanja elementov, kjer lahko vidimo, da elemente urejamo nenaraščajoče, torej elemente večje od pivota dajemo na desno stran tabele, manjše na levo in za enake nam je vseeno. Vrstici 6 in 18 ob predpostavki, da sta ustrezna pogoja izpolnjena pokličeta funkcijo 3.2.2: CONSIDER-COIN, kjer je $coins[r+1]$ eden izmed parametrov. Največja razlika je v vrstici 19, kjer smo nadomestili pogoj STOP-CONDITION in ob izpolnjenem pogoju lahko končamo izvajanje te veje rekurzije.

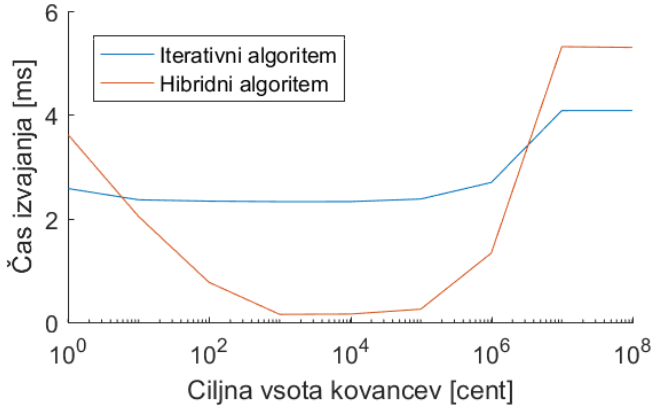
3.2.4 Eksperimentalna primerjava algoritmov

Za preverjanje učinkovitosti hibridnega algoritma smo naredili eksperiment s 30 000 kovanci. Odločili smo se, da bomo izbrali evrske vrednosti kovancev ter, da jih bomo hranili v centih. Vsak od kovancev je tako imel naključno vrednost iz množice $\{1, 2, 5, 10, 20, 50, 100, 200\}$.

Čase izvajanja algoritmov smo merili v odvisnosti od željenega zneska. Za prvi poskus, predstavljen z grafom na sliki 3.1 smo se odločili, da bo prvi

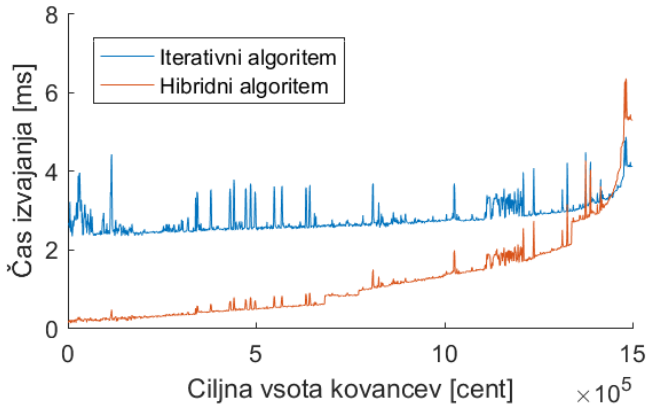
Function 3.2.3

```
1:  HYBRIDIZED-COIN-CHANGING(coins, left, right)
2:      if right < left then return ∅
3:      pivot ← coins[left]
4:      if right == left then
5:          result ← ∅
6:          CONSIDER-COIN(result, pivot)
7:          return result
8:      l ← left
9:      r ← right
10:     while l ≤ r do
11:         while coins[l] > pivot do l ← l + 1
12:         while coins[r] < pivot do r ← r − 1
13:         if l > r then break
14:         SWAP(coins, l, r)
15:         l ← l + 1
16:         r ← r − 1
17:     leftsol ← HYBRIDIZED-COIN-CHANGING(coins, left, r)
18:     if l − r == 2 then CONSIDER-COIN(leftsol, coins[r + 1])
19:     if goal ≤ 0 then return leftsol
20:     rightsol ← HYBRIDIZED-COIN-CHANGING(coins, l, right)
21:     return leftsol ∪ rightsol
```



Slika 3.1: Čas izvajanja algoritmov za reševanje problema menjava kovancev

znesek znašal en cent, nato pa ga bomo po vsaki iteraciji pomnožili z deset.



Slika 3.2: Čas izvajanja algoritmov za reševanje problema menjava kovancev

Pri drugem poskusu predstavljenem z grafom na sliki 3.2 smo začeli z zneskom desetih evrov (1000 centov) in znesek v vsaki iteraciji povečali za deset evrov. To smo izvajali dokler nismo dosegli zneska 15 000 evrov.

Pri obeh poskusih smo vsakič, ko smo določili nov znesek, nato oba algoritma izvedli po tisočkrat in izmerili povprečen čas izvajanja tisočih iteracij ter ga predstavili na grafu. Ob tem smo v vsaki izmed tisočih iteracij ponovno generirali podatke pred izvedbo algoritmov, torej vrednosti kovancev.

Na sliki 3.1 lahko opazujemo čase izvajanja algoritmov za različne zneske.

Na začetku, ko je znesek majhen je hibridni algoritem počasen, saj mora urediti skoraj vse elemente, da pride do majhne vrednosti, ki je naš željen znesek. Vendar lahko vidimo, da se to spremeni že pri desetih centih. Nato je algoritem vedno hitrejši.

Če izberemo prevelik znesek, nam kovancev lahko tudi zmanjka. Na sliki 3.1 vidimo, da je začelo kovancev zmanjkovati, ko je naš željeni znesek znašal med 10 000 evri in 100 000 evri. To je razumljivo, saj s 30 000 kovanci, tudi če bi bili vsi po dva evra, lahko dosežemo največji znesek 60 000 evrov.

Še bolj natančno se meja, kjer je pričelo zmanjkovati kovancev, vidi na sliki 3.2. Podamo lahko grobo oceno za kritično mejo uspešnosti hibridnega algoritma na tem naboru testnih podatkov, ki je nekje med 14 000 in 15 000 evri. Kar pomeni, da bo za katerikoli željen znesek manjši od 14 000 evrov in večji od desetih centov hibridni algoritem na tem naboru podatkov deloval veliko hitreje od navadnega. Torej sklepamo, da je hibridni algoritem veliko hitrejši, dokler nam ne prične zmanjkovati kovancev. Takrat mora namreč hibridni algoritem urediti skoraj vse elemente, da doseže željen znesek in posledično se poveča njegov čas izvajanja.

Na sliki 3.2 opazimo tudi, da čas izvajanja iterativnega algoritma precej niha. Menimo, da je to zato, ker je čas ene točke na grafu, povprečje tisočih meritev, kjer smo vsakokrat ponovno generirali podatke, kar pomeni, da so bili podatki drugačni in lahko v eni iteraciji bolj urejeni kot v drugi. To posledično poveča čas algoritma hitrega urejanja. Našo domnevo potrdi dejstvo, da je pri enakih ciljnih zneskih in vrednostih kovancev prav tako povečan čas izvajanja hibridnega algoritma. Poleg tega so časi izvajanja v milisekundah, torej ni velike razlike med meritvami.

Podatke smo namenoma generirali vsakokrat ponovno, saj smo želeli pridobiti splošen čas izvajanja za določeno število kovancev in nek določeni znesek, neodvisno od tega kako so urejeni podatki.

Za učinkovitost moramo upoštevati tudi, da je znesek neko število, ki ne vsebuje enic in ni premajhno, saj mora v takih primerih algoritem urediti skoraj vse elemente, pri tem pa trpi hitrost izvajanja.

Na sliki 3.1 lahko vidimo, da je pri ciljnem znesku enega centa, hibridni algoritem v poprečju precej slabši kot iterativni. Iz tega opažanja sklepamo, da bi hibridni algoritem za zneske, ki se končajo na enko, v povprečju lahko deloval počasneje od iterativnega, saj mora v takih primerih algoritem zopet urediti skoraj vse elemente.

Torej hibridni algoritem bo za ta problem učinkovit, kadar bo znesek nekoliko manjši od vsote vrednosti vseh kovancev.

3.3 Minimalno vpeto drevo

3.3.1 Definicija problema

Problem minimalnega vpetega drevesa [8] se pogosto pojavi pri načrtovanju optimalnih (najpogosteje najcenejših) energetskih, prometnih in komunikacijskih omrežij mest, ki jih lahko predstavimo s povezanim, neusmerjenim grafom.

Mesta nam v grafu predstavljajo vozlišča, povezave pa so možne povezave med mesti. Vsaki povezavi določimo ceno, ki jo je potrebno plačati, če želimo priti iz enega mesta do drugega. Povezave želimo izbrati tako, da bodo vsa vozlišča (mesta) povezana med sabo. Kar pomeni, da mora v grafu obstajati vsaj ena pot med vsakimi dvemi vozlišči. Tako lahko ta problem definiramo kot iskanje najcenejšega podgrafa, ki je povezan in acikličen. Tak podgraf imenujemo minimalno vpeto drevo.

Naj bo $G = (V, E)$ [3] povezan, neusmerjen graf, kjer je V množica vozlišč, E pa množica povezav grafa G. Za vsako povezavo $(u, v) \in E$ definirajmo ceno $w(u, v)$, ki je potrebna, da lahko povežemo vozlišči u in v.

Torej pri problemu minimalnega vpetega drevesa za nek graf G želimo poiskati aciklično podmnožico povezav $T \subseteq E$, ki med sabo poveže vsa vozlišča V in katere celotna cena $w(T)$ je najmanjša, torej velja enačba:

$$w(T) = \sum_{(u,v)\in T} w(u, v).$$

3.3.2 Požrešni algoritem

Poznamo več požrešnih algoritmov za reševanje problema minimalnega vpetega drevesa. Med najbolj znanimi so Primov, Kruskalov in algoritem Boruvke. Mi se bomo osredotočili na Kruskalov algoritem, ker edini vsebuje urejanje elementov, ki je ključno za hibridizacijo algoritma.

S tem algoritmom lahko optimalno rešimo problem, vendar moramo povezave E grafa G najprej urediti po naraščajoči ceni.

Algoritem prejme graf G in seznam cen povezav w. Nato pa v vsaki iteraciji izbere najcenejšo povezavo in jo doda v množico *result*.

Disjunktne množice, ki jih pri tem uporablja, implementiramo kot je prikazano v razdelku 2.3.

Vsaka disjunktna množica vsebuje vozlišča enega drevesa v gozdu. Funkcija 2.3.2: FIND-SET(u) pa nam vrne predstavnika drevesa, ki vsebuje u. Ko želimo dodati najcenejšo povezavo v rešitev, moramo najprej preveriti, da s tem ne bomo generirali cikla. To naredimo tako, da preverimo, ali vozlišči pripadata istemu drevesu. Torej, če velja FIND-SET(u) = FIND-SET(v), sta vozlišči že v istem drevesu, kar pomeni, da bi z dodajanjem povezave generirali cikel. Za združevanje dveh dreves pa Kruskalov algoritem uporablja funkcijo 2.3.3: UNION(u, v).

Function 3.3.1

```
1:  MST-KRUSKAL((V, E), w)
2:      result ← ∅
3:      for each vertex v ∈ V do
4:          MAKE-SET(v)
5:      sort the edges of E into nondecreasing order by weight w
6:      for each edge (u, v) ∈ E do
7:          if FIND-SET(u) ≠ FIND-SET(v) then
8:              result ← result ∪ {(u, v)}
9:              UNION(u, v)
10:     return result
```

Funkcija 3.3.1 je Kruskalov algoritem in najprej incializira množico povezav na prazno množico, nato pa se zanka *for* sprehodi čez vsa vozlišča in za vsako kreira drevo, ki vsebuje le vozlišče v. Tako dobimo gozd dreves, ki so disjunktna med seboj. Nato se v vrsticah 6-9 zanka sprehodi po povezavah, ki so urejene nepadajoče po cenah. V vsaki iteraciji se najprej preveri ali sta vozlišči povezave v istem drevesu, kar preverimo v vrstici 7, in če nista, se povezava doda v množico *result* v vrstici 8. Ob tem moramo tudi združiti drevesi za kar poskrbi vrstica 9.

Časovna zahtevnost Kruskalovega algoritma je odvisna od implementacije disjunktnih množic. Če predpostavimo uporabo hitrega urejanja in implementacijo disjunktnih množic na način predstavljen v razdelku 2.3, je časovna zahtevnost algoritma $O(|E| \log |E|)$. Ker pa velja $|E| < |V|^2$ in torej tudi $\log |E| = O(\log |V|)$, lahko časovno zahtevnost zapišemo kot $O(|E| \log |V|)$.

3.3.3 Hibridni požrešni algoritem

Tudi tokrat hibridni algoritem natančno sledi splošnemu algoritmu 2.4.2.

Malo večja sprememba, ki jo prinese hibridizacija, je uvedba nove funkcije 3.3.2: INITIALIZE-FOREST. Ko pričnemo pisat hibridni algoritem, hitro ugotovimo, da se vse izvaja rekurzivno, toda Kruskalov algoritem na začetku vsebuje incializacijo večih disjunktnih množic. Za to poskrbi funkcija 3.3.2: INITIALIZE-FOREST, tako da za vsako vozlišče $v \in V$ ustvari svojo množico oz. drevo, ki vsebuje le vozlišče v.

Function 3.3.2

```
1: INITIALIZE-FOREST(V)
2:     for each vertex v ∈ V do
3:         MAKE-SET(v)
```

Druga posebnost hibridnega algoritma je neuporaba spremenljivke *greedy-variables*, saj za potrebe razumevanja algoritma ni potrebna. Vendar jo v dejanski implementaciji algoritma potrebujemo za hranjenje disjunktnih množic oz. dreves, na podlagi katerih se odločamo ali bomo določeno pove-

zavo dodali v rešitev ali ne. Tako je spremenljivka *greedy_variables* inicializirana s pomočjo funkcije 2.3.1: MAKE-SET, uporablja pa se v funkciji 2.3.2: FIND-SET in funkciji 2.3.3: UNION, ki smo jih definirali v razdelku 2.3.

Zaradi razumljivosti algoritma smo zopet zamenjali generično ime *elements*, in sicer z imenom E, ki predstavlja vse povezave grafa. Te so kandidati za rešitev, torej za tvorbo minimalnega vpetega drevesa.

Kruskalov algoritem lahko ustavimo, če je v rešitvi, torej minimalnem vpetem drevesu, $|V| - 1$ povezav. To je naš pogoj STOP-CONDITION za izhod iz rekurzivne veje hibridne funkcije.

Function 3.3.3

```
1: CONSIDER-MST(result, (u, v))
2:     if FIND-SET(u) ≠ FIND-SET(v) then
3:         UNION(u, v)
4:         result ← result ∪ {(u, v)}
```

Funkcija 3.3.3: CONSIDER-MST zopet posnema splošno funkciji 2.4.1: CONSIDER in skupaj s funkcijo 3.3.2: INITIALIZE-FOREST tvori požrešni del algoritma. Kot parameter prejme kazalec na rešitev *result* in povezavo (u, v), ki povezuje vozlišči u in v. Če je pogoj v vrstici 2 izpolnjen, ta povezava sestavlja minimalno vpeto drevo.

Množica povezav E in indeksa $left$ in $right$ so parametri rekurzivne hibridne funkcije 3.3.4. V 3. vrstici hibridne funkcije lahko vidimo, da pivot postane povezava, ki je shranjena na indeksu $left$ v množici povezav E. V 11. in 12. vrstici opazimo, da urejamo povezave v nepadajočem vrstnem redu glede na ceno w. Če sta indeksa l in r oddaljena ravno za dve mesti pomeni, da je vmesni element na indeksu $r + 1$ trenutni kandidat za rešitev, saj ga v nasprotnem primeru rekurzivna funkcija nikoli več ne doseže. Tako vrstica 18 pokliče funkcijo 3.3.3: CONSIDER-MST s parametrom $E[r + 1]$. Če je pogoj v 19. vrstici izpolnjen pomeni, da smo že pridobili minimalno vpeto drevo, torej je število povezav v rešitvi $leftsol$ ravno $|V| - 1$.

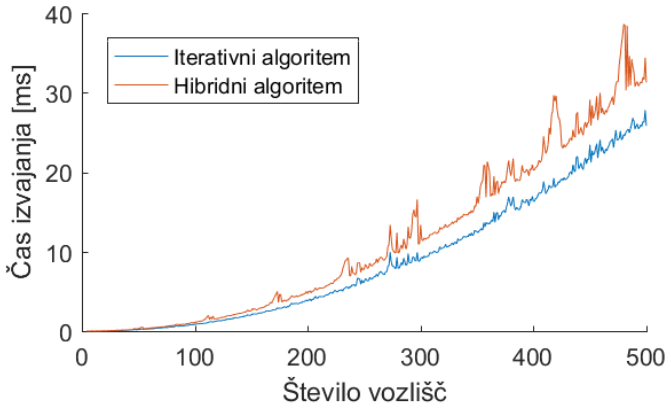
Function 3.3.4

```
1:  HYBRIDIZED-MST-KRUSKAL(E, left, right)
2:      if right < left then return ∅
3:      pivot ← E[left]
4:      if right == left then
5:          result ← ∅
6:          CONSIDER-MST(result, pivot)
7:          return result
8:      l ← left
9:      r ← right
10:     while l ≤ r do
11:         while w(E[l]) < w(pivot) do l ← l + 1
12:         while w(E[r]) > w(pivot) do r ← r − 1
13:         if l > r then break
14:         SWAP(E, l, r)
15:         l ← l + 1
16:         r ← r − 1
17:     leftsol ← HYBRIDIZED-MST-KRUSKAL(E, left, r)
18:     if l − r == 2 then CONSIDER-MST(leftsol, E[r + 1])
19:     if length(leftsol) ≥ length(V) − 1 then return leftsol
20:     rightsol ← HYBRIDIZED-MST-KRUSKAL(E, l, right)
21:     return leftsol ∪ rightsol
```

3.3.4 Eksperimentalna primerjava algoritmov

Učinkovitost delovanja našega hibridnega algoritma za problem minimalnega vpetega drevesa smo preverili na polnih grafih. Napisali smo program, ki je generiral grafe, ki smo jih uporabili za testiranje hibridnega in navadnega požrešnega algoritma. Ti so imeli od 4 do 500 vozlišč $|V|$.

Za vsak graf posebej smo določili cene povezav, ki so bile generirane naključno in so znašale med ena in številom povezav $|E|$. Za vsako število vozlišč $|V|$, smo algoritma izvajali tisočkrat, nato pa zabeležili povprečni čas izvajanja v milisekundah. Pri tem smo v vsaki iteraciji tudi vsakič ponovno generirali naključen poln graf z vozlišči V. Čase izvajanja smo nato predstavili z grafom.



Slika 3.3: Čas izvajanja algoritmov za iskanje minimalnega vpetega drevesa

Rezultate poskusa, torej graf izvajanja, lahko vidimo na sliki 3.3. Izvajanje hibridnega algoritma je na grafu predstavljeno z rdečo barvo, navadnega pa z modro. V nasprotju z našimi pričakovanji opazimo, da je hibridni algoritem minimalno vpeto drevo iskal dlje kot navadni požrešni algoritem.

Ker hibridni algoritem vsebuje zaustavitveni pogoj, smo pričakovali hitrejše izvajanje pri njem kot pri iterativni različici. Podobno kot je bil hitrejši hibridni algoritem za problem menjave kovancev.

Menimo, da bi k počasnosti, ki je nismo pričakovali, lahko pripomoglo dejstvo, da mora Kruskalov algoritem v povprečju pregledati veliko več po-

vezav, kot jih dejansko sestavlja rešitev. Majhna cena povezave sicer lahko predstavlja dobrega kandidata, ker bo tako drevo imelo nizko ceno, vendar ne moremo predvideti ali bo ta povezava generirala cikel ali ne. Zaradi dejstva da moramo skoraj vedno pregledati več povezav, kot pa jih je v rešitvi, sklepamo, da bi hibridni algoritem lahko deloval počasneje od iterativnega. Če upoštevamo še, da je čas izvajanja hibridnega rekurzivnega algoritma brez zaustavitvenega pogoja tudi počasnejši od iterativnega, lahko sklepamo, da bo hibridni algoritem deloval počasneje. Razlog je v tem, da mora urediti kar nakaj elementov, ob tem pa je hibridna rekurzivna struktura tudi bolj kompleksna od algoritma hitrega urejanja, kar pomeni, da bo hibridni algoritem potreboval dlje časa za pisanje na sklad in podobno.

Glede na to da smo merili povprečen čas tisočih iteracij različno urejenih povezav z naključnimi cenami, je možno, da so bili časi izvajanja algoritmov tudi zelo različni. Tako lahko samo rečemo, da je v povprečju algoritem deloval slabše, kar pomeni, da je za naključne polne grafe najbrž potrebno pregledati precejšnje število povezav, da pridemo do minimalnega vpetega drevesa.

Tudi pri tem algoritmu opazimo, da časa izvajanja algoritmov precej nihata. Zopet je čas ene točke na grafu, povprečje tisočih meritev, kjer smo vsakokrat ponovno generirali povezave in cene povezav, kar pomeni, da so bili podatki drugačni in lahko v eni iteraciji bolj urejeni kot v drugi. To posledično lahko poveča ali zmanjša čas algoritma hitrega urejanja in hibridnega algoritma.

3.4 Preprosti problem nahrbtnika

3.4.1 Definicija problema

Problem nahrbtnika [9] je problem, ki ga pogosto srečamo v vsakdanjem življenju. Pojavi se, ko gremo npr. na potovanje in želimo spakirati najbolj vredne in uporabne predmete, pri tem pa ne želimo preobremeniti kovčka. Podoben primer se zgodi, ko gremo v trgovino in želimo napolniti nakupo-

valno vrečko in nato zložiti predmete v avto.

Torej pri problemu nahrbtnika [10] želimo najti najbolj vredno podmnožico predmetov, ki ne presega volumna W. Če predmet lahko samo vzamemo ali pustimo, potem je to problem 0/1 nahrbtnika. Mi se bomo osredotočili na preprosti problem nahrbtnika, kjer lahko vzamemo predmet tudi le deloma.

Recimo, da imamo množico predmetov $S = \{1, 2, ..., n\}$, ki jih želimo zložiti v nahrbtnik, ki ima prostornino W. Za vsak predmet i definirajmo vrednost v_i, kjer $v_i \geq 0$, in velikost w_i, kjer $0 \leq w_i \leq W$.

Ker lahko vzamemo tudi le del predmeta, definirajmo x_i, ki je količina vsakega predmeta i, ki smo ga dodali v nahrbtnik. Pri tem pa mora za vsak $i \in S$ veljati $0 \leq x_i \leq w_i$ in količina vseh vzetih predmetov mora biti manjša ali enaka W. Torej:

$$\sum_{i \in S} x_i \leq W.$$

Celotno vrednost nahrbtnika, ki mora biti največja možna, pa izračunamo po formuli:

$$\sum_{i \in S} v_i(x_i/w_i).$$

3.4.2 Požrešni algoritem

Problem preprostega nahrbtnika lahko učinkovito rešimo s pomočjo požrešnega algoritma, vendar moramo predmete pred tem urediti v nenaraščajočem vrstnem redu glede na vrednost na enoto volumna v/w.

Požrešni algoritem (funkcija 3.4.1) prejme množico predmetov, kjer ima vsak svojo vrednost v in velikost w. Algoritem prejme tudi kapaciteto nahrbtnika W. Nato pa v vsaki iteraciji izbere trenutno najboljši predmet in njegovo velikost w doda v x.

Funkcija 3.4.1 v zanki *for* v vrsticah 3-4 najprej za vsak element i izračuna njegovo vrednost na enoto volumna v_i/w_i in incializira količine izbranih predmetov x_i na 0. Nato elemente uredi v nenaraščajočem vrstnem redu glede na v/w.

Function 3.4.1

```
1: GREEDY-FRACTIONAL-KNAPSACK(v, w, W)
2:     n ← lenghth(v)
3:     for i ← 1 . . . n do
4:         x[i] ← 0
5:     sort the items into nonincreasing order by v/w
6:     while W > 0 do
7:         x_i ← min{w_i, W}
8:         W ← W − x_i
9:     return x
```

Zanko *while* v vrsticah 6-8 izvajamo, dokler ne zapolnimo nahrbtnika. V vsaki iteraciji nato dodamo v x_i velikost elementa, ki ima največjo vrednost na enoto volumna v_i/w_i in volumen W zmanjšamo za količino dodanega elementa x_i.

V zadnji iteraciji pa se lahko zgodi, da ima trenutni element i večji ali enak volumen w_i kot je kapaciteta nahrbtnika W, torej $w_i \geq W$. Tako namesto celotne količine elementa v x_i shranimo le del, in sicer kar količino W. Algoritem nato vrne x, torej količine posameznih elementov v nahrbtniku.

Časovna zahtevnost požrešnega algoritma bi bila $O(n)$, če v algoritmu ne bi urejali elementov, ampak bi ti že bili predhodno urejeni. Vendar je velikokrat treba elemente še urediti, kar lahko najhitreje naredimo v času $O(n \log n)$, zato ima tudi požrešni algoritem časovno zahtevnost $O(n \log n)$.

3.4.3 Hibridni požrešni algoritem

Hibridni algoritem za problem preprostega nahrbtnika zopet precej natančno posnema generični algoritem 2.4.2.

V hibridnem algoritmu smo zaradi večje jasnosti algoritma definirali strukturo *items*, ki predstavlja predmete in ima za vsak predmet shranjeno vrednost v in velikost w. Le ta nadomešča elemente *elements* v generičnem algoritmu. V hibridnem algoritmu smo preimenovali globalno spremenljivko

greedy_variables v W. Spremenljivka sicer predstavlja kapaciteto oz. volumen nahrbtnika, ki ga želimo napolniti s čim vrednejšimi predmeti. Redefinirali smo tudi pogoj STOP-CONDITION, ki nam sedaj konča izvajanje rekurzije, če smo zapolnili nahrbtnik, torej če je njegov volumen enak 0.

Function 3.4.2

```
1: CONSIDER-ITEM(result, item_weight)
2:     if W > 0 then
3:         result ← result ∪ min{item_weight, W}
4:         W ← W − min{item_weight, W}
```

V funkciji 3.4.2: CONSIDER-ITEM s parametrom *item_weight*, ki predstavlja volumen opazovanega predmeta, najprej preverimo, če nismo zapolnili kapacitete nahrbtnika. Ob izpolnjenem pogoju v rešitev *result* dodamo ustrezno količino in posodobimo volumen nahrbtnika W.

Zaradi podobnosti s splošnim algoritmom 2.4.2 bomo pri opisu algoritma 3.4.3, omenili samo kakšno malenkost. Prva je v vrstici 6, ko v funkcijo CONSIDER-ITEM kot parameter vstopi volumen pivotnega elementa $pivot.w$ in ne kar sam *pivot*. V vrsticah 11 in 12 lahko vidimo, da predmete urejamo v nenaraščajočem vrstnem redu glede na vrednost na enoto volumna. Zopet sta posebni tudi vrstici 18 in 19, kjer v vrtici 19 lahko vidimo izstopni pogoj, v vrstici 18 pa kličemo funkcijo CONSIDER-ITEM s parametrom $items[r+1].w$, kjer hranimo volumen predmeta, ki je v *items* na mestu $r + 1$.

3.4.4 Eksperimentalna primerjava algoritmov

Delovanje hibridnega in navadnega požrešnega algoritma za preprosti problem nahrbtnika smo preverjali na množici 100 000 predmetov. Vrednost predmetov v je bila naključno generirano število med ena in tisoč. Prav tako smo tudi velikost predmetov oz. njihov volumen določili z naključnim številom vrednosti med ena in tisoč.

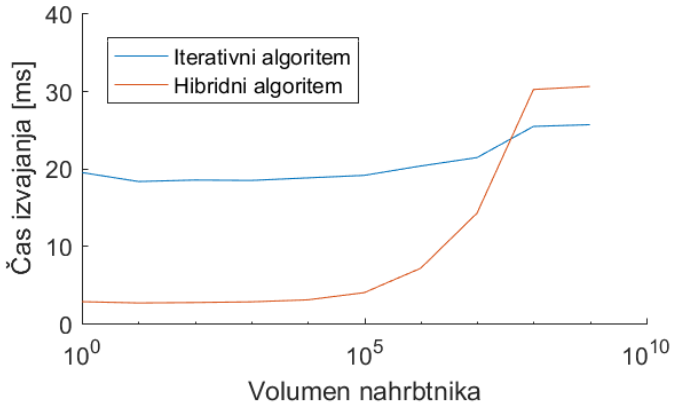
Za vsak volumen nahrbtnika W smo oba algoritma izvedli tisočkrat, nato smo izračunali povprečna časa izvajanja algoritmov za volumen W. Pri tem

Function 3.4.3

```
1:  HYBRIDIZED-FRACTIONAL-KNAPSACK(items(v, w), left, right)
2:      if right < left then return ∅
3:      pivot ← items[left]
4:      if right == left then
5:          result ← ∅
6:          CONSIDER-ITEM(result, pivot.w)
7:          return result
8:      l ← left
9:      r ← right
10:     while l ≤ r do
11:         while items[l].v/items[l].w > pivot.v/pivot.w do l ← l + 1
12:         while items[r].v/items[r].w < pivot.v/pivot.w do  r ← r − 1
13:         if l > r then break
14:         SWAP(items, l, r)
15:         l ← l + 1
16:         r ← r − 1
17:     leftsol ← HYBRIDIZED-FRACTIONAL-KNAPSACK(items, left, r)
18:     if l − r == 2 then CONSIDER-ITEM(leftsol, items[r + 1].w)
19:     if W ≤ 0 then return leftsol
20:     rightsol ← HYBRIDIZED-FRACTIONAL-KNAPSACK(items, l, right)
21:     return leftsol ∪ rightsol
```

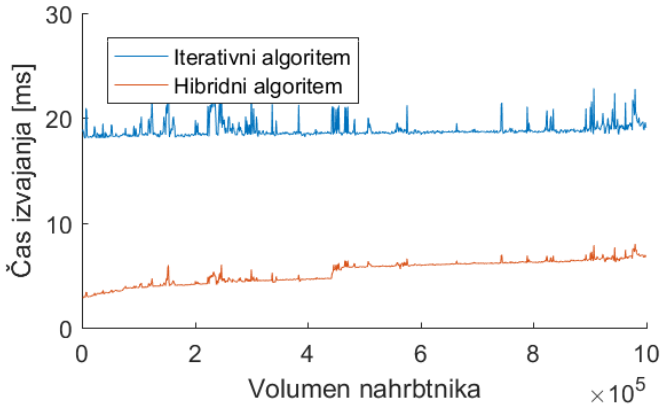
smo tudi vsakič ponovno generirali podatke za vse predmete, ker smo želeli čim bolj splošen rezultat.

Učinkovitost delovanja našega hibridnega algoritma smo merili glede na volumen nahrbtnika.



Slika 3.4: Čas izvajanja algoritmov za reševanje preprostega problema nahrbtnika

Tako smo si za volumen na sliki 3.4 izbrali najprej enico, nato pa smo volumen povečevali tako, da smo ga vsakokrat pomnožili z deset.



Slika 3.5: Čas izvajanja algoritmov za reševanje preprostega problema nahrbtnika

Na sliki 3.5, ki ponazarja graf, pa smo začeli z volumnom tisoč, nato pa

ga vsakič povečali za tisoč, dokler nismo dosegli volumna velikosti 1 000 000.

Iz grafa na sliki 3.5 lahko vidimo, da se čas izvajanja hibridnega algoritma v povprečju malenkostno povečuje, vendar je kljub temu precej hitrejši od navadnega. Ker je volumen nahrbtnika na začetku majhen, ga lahko z nekaj elementi hitro zapolnimo ter tako končamo izvajanje hibridnega algoritma. Ob tem urejamo zelo malo predmetov, kar pripomore k hitrosti.

Na sliki 3.4 lahko opazimo, da je hibridni algoritem nekaj časa veliko hitrejši od iterativnega. Ko pa se volumen iz 10 000 000 poveča na 100 000 000, se čas izvajanja hibridnega algoritma drastično poveča. Za večje volumne je hibridni algoritem počasnejši, ker se velikost nahrbtnika približuje vsoti volumnov predmetov oziroma jo celo preseže. Hibridni algoritem se mora tako izvesti do konca, torej urediti vse predmete, kar ga upočasni.

Na sliki 3.5 opazimo tudi, da povprečna časa izvajanja algoritmov malo nihata. To je posledica naše zasnove testa, saj je čas ene točke na grafu, povprečje tisočih meritev, kjer smo vsakokrat ponovno generirali podatke o predmetih, volumen nahrbtnika in število predmetov pa smo pustili pri miru.

Čas izvajanja hibridnega algoritma bi lahko še pohitrili z uvedbo dodatnega zaustavitvenega pogoja, kjer bi preverili, če je velikost pivota $pivot.w$ večja ali enaka volumnu nahrbtnika W, torej pogoj bi preverjal, če velja $pivot.w \geq W$. Če je ta pogoj izpolnjen smo tako našli še zadnji element, ki ga lahko dodamo v nahrbtnik. Posledično ni potrebno izvesti desnega dela rekurzije, kar malenkostno pohitri čas izvajanja hibridnega algoritma.

Torej iz poskusa lahko sklepamo, da se hibridni algoritem precej bolje obnese od navadnega, ko je volumen nahrbtnika nekoliko manjši od vsote volumnov predmetov.

3.5 Problem izbora aktivnosti

3.5.1 Definicija problema

Problem izbora aktivnosti [3] je problem, kjer bi radi razvrstili množico konkurenčnih aktivnosti, ki zahtevajo uporabo skupnega vira, tako da bomo

izbrali največjo množico paroma kompatibilnih aktivnosti.

Recimo, da imamo množico aktivnosti $S = \{1, 2, ..., n\}$, ki vse želijo uporabljati isti vir, npr. predavalnico, kjer pa se lahko izvaja samo ena aktivnost naenkrat. Za vsako aktivnost i definiramo začetni čas s_i in končni čas f_i, za katera mora veljati: $0 \leq s_i < f_i < \infty$. Če bomo aktivnost i izbrali, se bo izvajala med časom s_i in f_i, torej na pol odprtem intervalu $[s_i, f_i)$.

Aktivnosti i in j sta kompatibilni, če se intervala $[s_i, f_i)$ in $[s_j, f_j)$ ne prekrivata. Torej, če velja $s_i \geq f_j$ ali $s_j \geq f_i$. Pri problemu izbora aktivnosti, želimo izbrati največjo podmnožico paroma kompatibilnih aktivnosti.

3.5.2 Požrešni algoritem

Problem izbora aktivnosti lahko optimalno rešimo s požrešno metodo, če pred izvedno požrešnega algoritma aktivnosti uredimo v nepadajočem vrstnem redu glede na končni čas aktivnsoti f.

Če so objekti predhodno urejeni, potem ima požrešni algoritem za problem izbora aktivnosti časovno zahtevnost $O(n)$.

Požrešni algoritem prejme množico aktivnosti, kjer ima vsaka svoj začetni čas s in končni čas f, nato pa v vsaki iteraciji izbere trenutno najboljšo aktivnost in jo doda v množico A (glej funkcijo 3.5.1). Za aktivnost definiramo kar strukturo, kjer imamo za vsako aktivnost shranjen začetni in končni čas.

V funkciji 3.5.1 indeks i določa zadnji dodan element v množico A. Ker so aktivnosti urejene naraščajoče po končnem času, je $activities[i].f$ vedno najkasnejši čas od katerekoli aktivnosti v množici A. Torej velja:

$$activities[i].f = \max\{activities[k].f : k \in A\}.$$

V vrstici 4 in 5 izberemo prvo aktivnost $activities[1]$, ki ima tudi najmanjši končni čas, saj smo aktivnosti uredili in jo dodamo v množico A. Indeks i nastavimo, da kaže na to aktivnost. Zanka *for* v vrsticah 6-9 vsako aktivnost $activities[j]$ poskuša dodati v množico A. Za kompatibilnost z ostalimi aktivnostmi je dovolj če preverimo, da je začetni čas $activities[j].s$ večji od končnega časa $activities[i].f$ zadnje dodane aktivnosti v množico A

Function 3.5.1

```
1:  GREEDY-ACTIVITY-SELECTOR(activities(s, f))
2:      sort the activities into nondecreasing order by finish time f
3:      n ←length(activities)
4:      A ← {activities[1]}
5:      i ← 1
6:      for j ← 2 . . . n do
7:          if activities[j].s ≥ activities[i].f then
8:              A ← A ∪ {activities[j]}
9:              i ← j
10:     return A
```

(vrstica 7). Če je aktivnost kompatibilna potem vrstici 8-9 dodata aktivnost $activities[j]$ v A in i nastavita na vrednost j.

Požrešni algoritem ima časovno zahtevnost enako algoritmu hitrega urejanja, torej $O(n \log n)$.

3.5.3 Hibridni požrešni algoritem

Tudi ta hibridni algoritem sledi splošnemu algoritmu, vendar nima zaustavitvenega pogoja STOP-CONDITION, saj iščemo največjo množico kompatibilnih aktivnosti. Lahko bi se zgodilo, da bi bila ravno zadnja aktivnost tudi kompatibilna s prejšnje dodanimi, kar pomeni, da rekurzivne funkcije ne moremo ustaviti dokler ne pregleda vseh aktivnosti. Posledično to pomeni, da se čas izvajanja v primerjavi z navadnim požrešnim algoritmom ne bo nič izboljšal, ampak celo poslabšal.

Skladno z drugimi algoritmi smo tudi pri tem algoritmu redefinirali spremenljivko *elements*, in sicer sedaj v funkcijo podajamo aktivnosti *activities*, kjer vsaka aktivnost hrani začetni in končni čas svojega izvajanja. Globalna spremenljivka *greedy_variables* pa nam pri tem algoritmu hrani končni čas zadnje dodane aktivnosti v rešitev, torej smo jo ustrezno preimenovali v $previous f$.

Function 3.5.2

```
1: CONSIDER-ACTIVITY(result, activity (s, f))
2:     if activity.s ≥ previousf then
3:         result ← result ∪ {activity (s, f)}
4:         previousf ← activity.f
```

Funkcija 3.5.2: CONSIDER-ACTIVITY kot parameter prejme aktivnost z začetnim in končnim časom, jo ob izpolnjenem pogoju doda v rešitev in posodobi končni čas $previousf$ na končni čas dodane aktivnosti $activity.f$.

V hibridni funkciji 3.5.3 lahko v vrsticah 11-12 vidimo, da urejamo končne čase od aktivnosti v nepadajočem vrstnem redu. V vrstici 18 poskusimo dodati v rešitev aktivnost iz mesta $r + 1$.

3.5.4 Eksperimentalna primerjava algoritmov

Pri problemu izbora aktivnosti iščemo največjo množico kompatibilnih aktivnosti. To za naš hibridni algoritem pomeni, da bo moral obiskati vse aktivnosti, saj bi ravno zadnja tudi lahko bila del rešitve, kar nas takoj napelje na to, da bo algoritem počasnejši od navadnega.

Hitrost algoritmov smo preverjali glede na število aktivnosti n. Za vsako aktivnost smo naključno določili začetni čas, ki je dobil vrednost iz intervala $[0, n)$. Končni čas pa smo izračunali tako, da smo začetnemu času prišteli naključno število iz intervala $[1, 100]$.

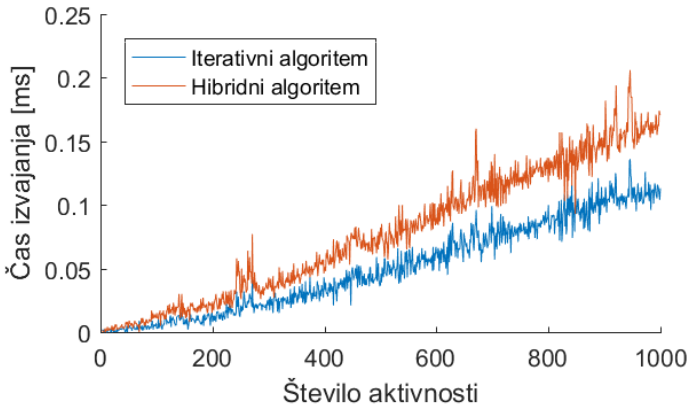
Za izračun hitrosti algoritmov smo najprej določili dve aktivnosti, nato pa smo to število povečevali do števila tisoč. Vsakič, ko smo povečali število aktivnosti, smo nato algoritma izvedli tisočkrat, pri tem pa v vsaki iteraciji ponovno generirali aktivnosti, za bolj splošen čas izvajanja. Tako smo dobili povprečen čas izvajanja obeh algoritmov za določeno število aktivnosti, ki smo ga tudi predstavili na sliki 3.6.

Na sliki 3.6 se lepo vidi, da je hibridni algoritem občutno počasnejši od navadnega požrešnega.

Oba algoritma sta si zelo podobna, vendar hibridni algoritem vsebuje

Function 3.5.3

```
1:  HYBRIDIZED-ACTIVITY-SELECTOR(activities (s, f), left, right)
2:      if right < left then return ∅
3:      pivot ← activities [left]
4:      if right == left then
5:          result ← ∅
6:          CONSIDER-ACTIVITY(result, pivot)
7:          return result
8:      l ← left
9:      r ← right
10:     while l ≤ r do
11:         while activities [l].f < pivot.f do l ← l + 1
12:         while activities [r].f > pivot.f do r ← r − 1
13:         if l > r then break
14:         SWAP(activities , l, r)
15:         l ← l + 1
16:         r ← r − 1
17:     leftsol ← HYBRIDIZED-ACTIVITY-SELECTOR(activities , left, r)
18:     if l − r == 2 then CONSIDER-ACTIVITY(leftsol, activities [r + 1])
19:     rightsol ← HYBRIDIZED-ACTIVITY-SELECTOR(activities , l, right)
20:     return leftsol ∪ rightsol
```



Slika 3.6: Čas izvajanja algoritmov za reševanje problema izbora aktivnosti

kompleksnejšo rekurzijo z več spremenljivkami, ki jih rekurzija shrani na sklad. Konstantno branje in pisanje s sklada bi lahko povečalo čas izvajanja hibridnega algoritma. Tudi hitro urejanje shranjuje podatke na sklad vendar jih ima manj. Tako bi čas hibridne funkcije lahko bil slabši zaradi naše implementacije.

Kot pri drugih algoritmih opazimo tudi, da meritve zelo nihajo, kar pa je posledica zasnove našega eksperimenta.

Na podlagi tega poskusa lahko sklepamo, da hibridizacija ne bo delovala pri algoritmih, kjer je potrebno pregledati vse elemente, ne le prvih nekaj, za pridobitev optimalne rešitve.

3.6 Razvrščanje poslov v delavnici z enim strojem

3.6.1 Definicija problema

Neka delavnica ima en sam stroj, ki obdeluje posle enega za drugim in za vsak posel porabi enoto časa [11]. Za vsak posel je določen rok izgotovitve in kazen, ki jo je treba plačati, če posel zamudi rok. Pri razvrščanju poslov iščemo podmnožico poslov in vrstni red njihovega izvajanja, s katerima

dosežemo najmanjšo skupno kazen neizgotovljenih poslov, pri pogoju, da so vsi posli podmnožice dokončani do svojega roka. Torej je vrstni red izvajanja samo permutacija poslov. Prvi posel se začne izvajti ob času 0 in se konča ob času 1, drugi posel se začne ob času 1 in konča ob času 2 in tako dalje.

Naj bo $S = \{1, 2, ..., n\}$ množica poslov velikosti n. Za vsak posel i definiramo rok izgotovitve d_i do katerega se mora končati posel i in za katerega mora veljati: $1 \leq d_i \leq n$. Definiramo tudi kazen p_i, kjer $p_i \leq 0$, ki jo bomo morali plačati, če se posel i, ne bo izvedel do roka. V nasprotnem primeru nam ne bo treba plačati nič.

Torej pri razvrščanju poslov [3] iščemo urnik izvajanja poslov, tako da bo skupna kazen izgotovljenih poslov najmanjša.

3.6.2 Požrešni algoritem

Požrešni algoritem nam optimalno reši razvrščanje poslov tako, da posle najprej uredi padajoče glede na kazen p.

Algoritem prejme množico poslov, kjer ima vsak svoj rok d in kazen p [12]. Nato v vsaki iteraciji izbere trenutno najboljši posel in ga, če je mogoče, doda v urnik. Algoritem učinkovito implementira disjunktne množice na način, ki je predstavljen v razdelku 2.3.

Torej imamo n možnih terminov in za vsakega kreiramo svojo disjunktno množico. Če sta termina i in j v isti množici potem je zadnji prost termin pred časom $t = i$ in $t = j$ kar enak za oba. Funkcija 2.3.2: FIND-SET(i) nam vrne predstavnika množice, ki vsebuje termin i. Vrednost, ki nam jo vrne $latest\_available$[FIND-SET(i)] je zadnji prost termin pred terminom, ki ga vrne funkcija 2.3.2: FIND-SET(i).

Vsakokrat, ko zapolnimo termin $[i-1, i]$ z nekim poslom, postane zadnji prost termin pred časom $i-1$ in i enak. Posodobimo ga lahko tako, da najprej združimo množici s termini, torej pokličemo funkcijo 2.3.3: UNION($i-1, i$), sledi posodobitev zadnjega prostega termina predstavnika množice i oz. $i-1$.

Pri funkciji 3.6.1 se zanka *for* v vrsticah 2-4 najprej sprehodi čez vse posle in za vsakega kreira množico s terminom i. Ob tem tudi inicializira

Function 3.6.1

```
1:  GREEDY-UNIT-TASK-SCHEDULER(d, p)
2:      for i ← 0 ... n do
3:          latest_available[i] ← i
4:          MAKE-SET(i)
5:      sort the tasks into nonincreasing order by penalty p
6:      for i ← 1 ... n do
7:          j ← latest_available[FIND-SET(d_i)]
8:          if j ≠ 0 then
9:              schedule task i at time interval [j − 1, j]
10:             old_available ← latest_available [FIND-SET(j − 1)]
11:             UNION(j − 1, j)
12:             latest_available[FIND-SET(j)] ← old_available
13:     return schedule
```

zadnji prost termin. Če je zadnji prost termin enak 0, pomeni, da posla ne moremo uvrstiti na urnik, brez da bi morali plačati kazen. Zanka *for* se nato v vrsticah 6-12 sprehodi čez urejene posle in v vsaki iteraciji poskuša dodati posel v urnik. V vrstici 7 poiščemo zadnji prost termin za posel, ki ima rok izgotovitve d_i. Če ta ni enak 0 (vrstica 8), ga v vrstici 9 dodamo v urnik. V vrsticah 10-11 združimo množici, ki imata rok izgotovitve $j - 1$ in j in posodobimo zadnji prost termin od predstavnika množice, ki sedaj vsebuje termina $j - 1$ in j.

Časovna zahtevnost algoritma je zopet odvisna od implementacije disjunktnih množic. Če poleg uporabe urejanja predpostavimo implementacijo disjunktnih množic iz razdelka 2.3, je časovna zahtevnost algoritma $O(n \log n)$.

3.6.3 Hibridni požrešni algoritem

Požrešni algoritem za razvrščanje poslov tako kot Kruskalov uporablja disjunktne množice. Tako moramo tudi pri tem hibridnem algoritmu uvesti

novo funkcijo 3.6.2: INITIALIZE-SET, kjer ustvarimo n disjunktnih množic možnih terminov za posel.

Te disjunktne množice in spremenljivka *latest_available* nadomestijo v generičnem algoritmu spremenljivko *greedy_variables*. Disjunktne množice zopet ustvarimo s funkcijo 3.6.2: MAKE-SET in jih uporabljamo v funkciji 2.3.2: FIND-SET in 2.3.3: UNION.

Function 3.6.2

```
1: INITIALIZE-SET(tasks)
2:     for i ← 0 ... n do
3:         MAKE-SET(i)
```

Kot pri ostalih algoritmih spremenljivko *elements* ustrezno preimenujemo v *tasks*, kjer imamo za vsak posel *task* shranjen njegov rok izgotovitve in kazen.

Ta hibridni algoritem, tako kot hibridni algoritem za problem izbora aktivnosti, nima zaustavitvenega pogoja STOP-CONDITION, saj želimo čim večje število poslov uvrstiti v urnik in zmanjšati skupno kazen.

Function 3.6.3

```
1: CONSIDER-TASK(result, task (d, p))
2:     j ← latest_available[FIND-SET(task .d)]
3:     if j ≠ 0 then
4:         result ← result ∪ {task }
5:         old_available ← latest_available[FIND-SET(j − 1)]
6:         UNION(j − 1, j)
7:         latest_available[FIND-SET(j)]← old_available
```

V funkciji 3.6.3: CONSIDER-TASK, ki prejme posel *task*, lahko vidimo požrešni del hibridnega algoritma.

Hibridna funkcija 3.6.4 je podobna splošni iz razdelka 2.4. Opazimo lahko, da kazni poslov urejamo v nenaraščajočem vrstnem redu in da v vrstici 18 poskusimo dodati v rešitev posel iz mesta $r + 1$.

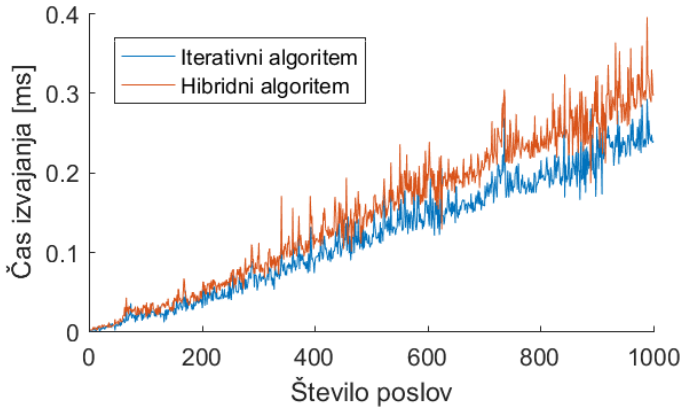
Function 3.6.4

```
1:  HYBRIDIZED-TASK-SCHEDULER(tasks(d, p), left, right)
2:      if right < left then return ∅
3:      pivot ← tasks[left]
4:      if right == left then
5:          result ← ∅
6:          CONSIDER-TASK(result, pivot)
7:          return result
8:      l ← left
9:      r ← right
10:     while l ≤ r do
11:         while tasks[l].p > pivot.p do l ← l + 1
12:         while tasks[r].p < pivot.p do r ← r − 1
13:         if l > r then break
14:         SWAP(tasks, l, r)
15:         l ← l + 1
16:         r ← r − 1
17:     leftsol ← HYBRIDIZED-TASK-SCHEDULER(tasks, left, r)
18:     if l − r == 2 then CONSIDER-TASK(leftsol, tasks[r + 1])
19:     rightsol ← HYBRIDIZED-TASK-SCHEDULER(tasks, l, right)
20:     return leftsol ∪ rightsol
```

3.6.4 Eksperimentalna primerjava algoritmov

Naredili smo eksperiment, da bi preverili učinkovitost delovanja našega hibridnega algoritma.

Za vsak posel smo za rok izgotovitve izbrali naključno število med ena in n, ki je število poslov. Določiti smo morali tudi kazen, ki je prav tako naključna, in sicer imajo naključne možne vrednosti razpon od ena do dvakratnega števila poslov.



Slika 3.7: Čas izvajanja algoritmov za razvrščanje poslov na stroju

Za računanje hitrosti algoritmov pa je za nas bolj pomembno število poslov. Zopet smo se odločili, da bomo eksperiment najprej izvedli z dvema posloma, nato pa to število povečevali do tisoč. Vsakokrat, ko smo povečali število poslov, smo nato algoritma izvedli tisočkrat, ob tem pa smo vedno tudi ponovno generirali vse podatke. Tako smo pridobili povprečen čas izvajanja splošnega problema z določenim številom poslov, ki smo ga prikazali tudi na sliki 3.7

Opazimo, da je hibridni algoritem počasnejši od navadnega in prisotno je precejšnje nihanje meritev. To je posledica ponovnega generiranja poslov v vsaki izmed tisočih iteracij, kar pomeni, da so bili podatki lahko različni in tudi različno dobro urejeni. Posledično to lahko spremeni čase izvajanja algoritmov. Ker smo za eno točko na grafu vzeli povprečje vseh tisočih iteracij je na grafu prisotno veliko šuma.

Oba algoritma uredita vse elemente, vendar so meritve pokazale, da je iterativni algoritem vseeno hitrejši. Hibridni algoritem vsebuje kompleksnejšo rekurzijo z več spremenljivkami, ki jih rekurzija shranjuje na sklad. Tako mora tekom izvajanja algoritma podatke konstantno brati in pisati na sklad, kar bi lahko povečalo čas izvajanja hibridnega algoritma. Tudi hitro urejanje uporablja sklad, vendar potrebuje manj podatkov za urejanje. Čas hibridne funkcije bi lahko bil slabši zaradi naše implementacije.

Zopet lahko sklepamo, da so slabši rezultati hibridnega algoritma, posledica pomanjkanja zaustavitvenega pogoja. Ta algoritem mora torej pregledati vse posle za pridobitev optimalnega urnika z najmanjšo kaznijo, kar pomeni, da je neprimeren za hibridizacijo.

Poglavje 4

Zaključek

V diplomskem delu smo preverjali učinkovitost hibridnega požrešnega algoritma v primerjavi z navadnim požrešnim algoritmom na določenem naboru podatkov. Za testno množico smo si izbrali pet požrešnih algoritmov, in sicer požrešni algoritem za problem menjave kovancev, Kruskalov algoritem, požrešni algoritem za preprosti problem nahrbtnika in izbora aktivnosti ter požrešni algoritem za razvrščanje poslov v delavnici z enim strojem.

Hibridni algoritem za problem razvrščanja kovancev se je izkazal kot učinkovit, saj je na testnih podatkih v primerjavi z navadnim požrešnim algoritmom dosegal veliko manjše čase izvajanja. Podobno je bila uspešna tudi hibridizacija požrešnega algoritma za preprosti problem nahrbtnika. Presentil pa nas je Kruskalov algoritem. Kljub temu, da se hibridni algoritem ni izvajal do konca, to ni pripomoglo k boljšemu času hibridnega algoritma. Hibridna požrešna algoritma za problem izbora aktivnosti in za razvrščanje poslov v delavnici z enim strojem po pričakovanjih nista dosegla boljšega časa izvajanja od svoje iterativne različice, saj moramo za pridobitev optimalne rešitve pregledati vse elemente.

Delovanje generičnega hibridnega algoritma bi bilo dobro preveriti, še na kakšnem drugem požrešnem algoritmu, kjer lahko pridobimo rešitev preden do konca uredimo elemente. V diplomski nalogi so bili vsi testni požrešni algoritmi optimalni, torej bi bil smiselen kandidat tudi kateri od aproksima-

cijskih požrešnih algoritmov.

Bolj natančno bi lahko določili pogoje oz. primere za uspešnost poljubnega hibridnega algoritma, če bi implementirali še več algoritmov, da bi lahko iz njih potegnili nek vzorec, ki je ključen za učinkovitost hibridnega algoritma.

Rezultati izvedenih poskusov so sicer spodbudni, vendar na podlagi njih ne moremo sklepati, da bo poljubni hibridni algoritem hitrejši samo zato, ker lahko pridobimo rešitev preden se urejanje elementov izvede do konca.

Literatura

[1] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., 2006.

[2] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2005.

[4] Jurij Mihelič. Quicksort s križanjem kazalcev. `http://lalg.fri.uni-lj.si/jurij/blog/quicksort-s-krizanjem-kazalcev/`. [Dostopano: 7. 9. 2017].

[5] Xuan Cai. Canonical coin systems for change-making problems. `https://arxiv.org/pdf/0809.0400.pdf`. [Dostopano: 5. 9. 2017].

[6] Jurij Mihelič. Požrešni algoritmi. `https://ucilnica1516.fri.uni-lj.si/pluginfile.php/27323/mod_resource/content/0/P18-Pozresni%20algoritmi.pdf`. [Dostopano: 30. 8. 2017].

[7] Kevin Wayne. Greedy algorithms 1. `https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf`. [Dostopano: 5. 9. 2017].

[8] Matija Lokar. Minimalno vpeto drevo. `https://wiki.lokar.fmf.uni-lj.si/r2wiki/index.php/Minimalno_vpeto_drevo`. [Dostopano: 30. 8. 2017].

[9] Knapsack problem. https://en.wikipedia.org/wiki/Knapsack_problem. [Dostopano: 30. 8. 2017].

[10] Michael T Goodrich and Roberto Tamassia. *Algorithm design: foundation, analysis and internet examples*. John Wiley & Sons, 2002.

[11] Boštjan Vilfan. *Osnovni algoritmi*. Fakulteta za računalništvo in informatiko, 2002.

[12] Siavosh Benabbas. Tutorial notes 2. http://www.cs.toronto.edu/~siavosh/csc373h/files/TN2.pdf. [Dostopano: 30. 8. 2017].