Конструкција и анализа алгоритама 2

Материјал са предавања

Миодраг Живковић
е-маил: ezivkovm@matf.bg.ac.rs
УРЛ: www.matf.bg.ac.rs/~ezivkovm

Весна Маринковић
е-маил: vesnap@matf.bg.ac.rs
УРЛ: www.matf.bg.ac.rs/~vesnap
Математички факултет, Београд

Аутори:
проф. др Миодраг Живковић, редовни професор Математичког факултета у Београду
др Весна Маринковић, доцент Математичког факултета у Београду

КОНСТРУКЦИЈА И АНАЛИЗА АЛГОРИТАМА 2

Глава 1

Предговор

Овај текст представља пратећи материјал за курс "Конструкција и анализа алгоритама 2" на Математичком факултету Универзитета у Београду. Текст је првенствено заснован на књизи "Алгоритми" Миодрага Живковића, али и на осталој препорученој литератури. Велику захвалност аутори дугују Николи Ајзенхамеру чија је скрипта за исти курс помогла у припреми неких тема из овог материјала. Аутори су такође захвални и студентима који су им обратили пажњу на грешке у материјалу. Скрипта покрива теме које се прелазе на часовима предавања, али ни у ком случају не може заменити похађање часова предавања.

Материјал је и даље у фази дораде те уколико уочите било какву грешку или пропуст, молимо вас да се јавите ауторима скрипте.

Аутори

Садржај

Глава 2

Геометријски алгоритми

2.1 Увод

Геометријски алгоритми играју важну улогу у многим областима, на пример у рачунарској графици, пројектовању помоћу рачунара, пројектовању VLSI (интегрисаних кола високе резолуције), роботици и базама података. У рачунарски генерисаној слици може бити на хиљаде или чак милионе тачака, линија, квадрата или кругова; пројектовање компјутерског чипа може да захтева рад са милионима елемената. Сви ови проблеми садрже обраду геометријских објеката. Пошто величина улаза за ове проблеме може бити врло велика, веома је значајно развити ефикасне алгоритме за њихово решавање.

Размотрићемо неколико основних геометријских алгоритама — оних који се могу користити као елементи за изградњу сложенијих алгоритама. Објекти са којима се ради су тачке, праве, дужи и многоуглови. Алгоритми обрађују ове објекте, односно израчунавају неке њихове карактеристике. Најпре ћемо дати основне дефиниције и навести структуре података погодне за представљање појединих објеката. **Тачка** p у равни представља се као пар координата (x, y) (претпоставља се да је изабран фиксирани координатни систем). **Права** је представљена паром тачака p и q (произвољне две различите тачке на правој), и означаваћемо је са $-p - q-$. **Дуж** се такође задаје паром тачака p и q које представљају крајеве те дужи, и означаваћемо је са $p - q$. **Пут** P је низ тачака p_1, $p_2, \ldots, p_k$ и дужи $p_1 - p_2$, $p_2 - p_3, \ldots, p_{k-1} - p_k$ које их повезују. Дужи које чине пут су његове **ивице** (странице). **Затворени пут** је пут чија се последња тачка поклапа са првом. Затворени пут зове се и **многоугао**. Тачке које дефинишу многоугао су његова **темена**. На пример, троугао је многоугао са три темена. Многоугао се представља низом, а не скупом тачака, јер је битан редослед којим се тачке задају (променом редоследа тачака из истог скупа у општем случају добија се други многоугао). **Прост многоугао** је онај код кога одговарајући пут нема пресека са самим собом; другим речима, једине ивице које имају заједничке тачке су суседне ивице са својим заједничким теменом. Прост многоугао ограничава једну област у равни, **унутрашњост** многоугла. **Конвексни многоугао** је многоугао чија унутрашњост са сваке две тачке које садржи, садржи и све тачке те

дужи. **Конвексни пут** је пут од тачака $p_1, p_2, \ldots, p_k$ такав да је многоугао $p_1 p_2 \ldots p_k, p_1$ конвексан.

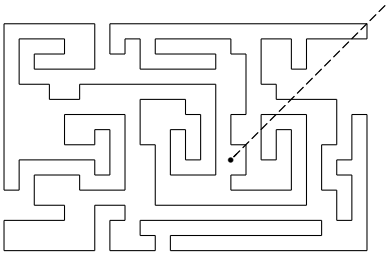
Претпоставља се да је читалац упознат са основама аналитичке геометрије. У алгоритмима које ћемо разматрати наилази се, на пример, на израчунавање пресечне тачке двеју дужи, утврђивање да ли две тачке леже са исте стране задате праве, израчунавање растојања између две тачке. Све ове операције могу се извести за време ограничено константом, константном применом основних аритметичких операција. При томе, на пример, претпостављамо да се квадратни корен може израчунати за константно време.

Честа неугодна карактеристика геометријских проблема је постојање многих "специјалних случајева". На пример, две праве у равни обично се секу у једној тачки, сем ако су паралелне или се поклапају. При решавању проблема са две праве, морају се предвидети сва три могућа случаја. Сложенији објекти проузрокују појаву много већег броја специјалних случајева, о којима треба водити рачуна. Обично се већина тих специјалних случајева решава непосредно, али потреба да се они узму у обзир чини понекад конструкцију геометријских алгоритама врло исцрпљујућом. Ми ћемо у овом поглављу често игнорисати детаље који нису од суштинског значаја за разумевање основних идеја алгоритама, али читаоцу саветујемо да размотри решавање сваког од специјалних случајева на који се при конструкцији алгоритма наиђе.

2.2 Утврђивање да ли задата тачка припада многоуглу

Разматрање започињемо једним једноставним проблемом.

Проблем. Задат је прост многоугао P и тачка q. Установити да ли је тачка у или ван многоугла P.



Слика 2.1: Утврђивање припадности тачке унутрашњости простог многоугла.

Проблем изгледа једноставно на први поглед, али ако се разматрају сложени неконвексни многоуглови, као онај на слици 2.1, проблем сигурно није једноставан. Први интуитивни приступ је покушати некако "изаћи напоље", полазећи од задате тачке. Посматрајмо произвољну полуправу са теменом q. Види се да је довољно пребројати пресеке са страницама многоугла, све до достизања спољашње области. У примеру на слици 2.1,

идући на североисток од дате тачке (пратећи испрекидану линију), наилазимо на шест пресека са многоуглом до достизања спољашње области. Истина, већ после четири пресека стиже се ван многоугла, али то рачунар "не види"; зато се броји укупан број пресека полуправе са страницама многоугла. Пошто нас последњи пресек пре изласка изводи из многоугла, а претпоследњи нас враћа у многоугао, итд., тачка је ван многоугла. Уопште тачка је у многоуглу ако и само ако је број пресека непаран (специјалне случајеве на тренутак занемарујемо).

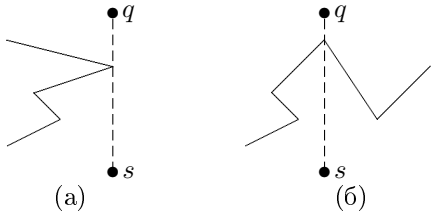
Алгоритам Tačka_u_mnogouglu_1(P, q); {први покушај}
Улаз: P (прост многоугао са теменима $p_1, p_2, \ldots, p_n$ и ивицама $e_1, e_2, \ldots, e_n$), и q (тачка).
Излаз: *Pripada* (Булова променљива, *true* акко q припада P).
begin
Изабрати произвољну тачку s ван многоугла;
Нека је L дуж $q - s$;
$broj := 0$;
for све странице e_i многугла **do**
if e_i сече L **then**
{претпостављамо да пресек није ни теме ни ивица, видети текст}
$broj := broj + 1$;
if $broj$ је непаран **then** $pripada := true$
else $pripada := false$
end

Као што је речено у претходном одељку, обично постоје неки специјални случајеви које треба посебно размотрити. Нека је s тачка ван многоугла, и нека је L дуж која спаја q и s. Циљ је утврдити да ли q припада унутрашњости многоугла P на основу броја пресека дужи L са ивицама многоугла P. Међутим, дуж L може се делом преклапати са неким ивицама многоугла P. Ово преклапање очигледно не треба бројати у пресеке. Први специјални случај је када дуж L садржи неко теме p_i многоугла. На слици 2.2(а) види се пример случаја кад пресек L са теменом не треба бројати, а на слици 2.2(б) пример кад тај пресек треба бројати. Постоји више начина за утврђивање да ли неки овакав пресек треба бројати или не.

- Један начин је следећи: ако су два темена многоугла суседна темену p_i и са исте стране L, пресек се не рачуна. У противном, ако су два суседна темена са различитих страна дужи L, пресек се броји.

- Други могући начин је следећи: с обзиром на то да разматрамо пресеке праве паралелне са y осом, за сваку ивицу многоугла рачунамо рецимо лево теме, а десно не (темена класификујемо као лево и десно у односу на вредности x координате). Тиме нећемо рачунати пресек налик оном на слици 2.2(а) јер је за обе ивице пресек кроз десно теме, а пресек као на слици 2.2(б) хоћемо једном, јер је за једну од ивица које се сустичу у том темену то лево теме, а за другу десно.

Други случај је када се неколико узастопних темена заједно са p_i налазе на полуправој, теме p_j претходи овим теменима, а теме p_k је прво после



Слика 2.2: Специјални случајеви кад права сече ивицу кроз теме.

њих на многоуглу: ако су темена p_i и p_j са различитих, односно са исте стране полуправе, пресек се рачуна, односно не рачуна.

Развијајући овај алгоритам, имплицитно смо претпостављали да радимо са сликом. Проблем је нешто другачији кад је улаз дат низом координата, што је уобичајено. На пример, кад посао радимо ручно и видимо многоугао, лако је наћи добар пут (онај са мало пресека) до неке тачке ван многоугла. У случају многоугла датог низом координата, то није лако. Највећи део времена троши се на израчунавање пресека. Тај део посла може се битно упростити ако је дуж $q - s$ паралелна једној од оса — на пример y-оси. Број пресека са овом специјалном дужи може бити много већи него са оптималном дужи (коју није лако одредити — читаоцу се оставља да размотри овај проблем), али је налажење пресека много једноставније (за константни фактор).

Алгоритам Tačka_u_mnogouglu_2(P, q); {други покушај}
Улаз: P (прост многоугао са теменима $p_1, p_2, \ldots, p_n$ и страницама $e_1, e_2, \ldots, e_n$), и q (тачка са координатама (x_0, y_0)).
Излаз: *Pripada* (булова променљива, *true* акко q припада P).
begin
    $Broj := 0$;
    for све гране e_i многоугла **do**
        if права $x = x_0$ сече e_i **then**
            {Претпостављамо да пресек није ни теме ни страница многоугла}
            Нека је y_i y-координата пресека праве $x = x_0$ са e_i;
            if $y_i < y_0$ **then** {пресек је испод q}
                $Broj := Broj + 1$;
    if *Broj* је непаран **then** $Pripada := true$
    else $Pripada := false$
end

Сложеност. За израчунавање пресека две дужи потребно је константно време. Нека је n број ивица многоугла. Кроз основну петљу алгоритма пролази се n пута. У сваком проласку налази се пресек две дужи и извршавају се још неке операције за константно време. Дакле, укупно време извршавања алгоритма је $O(n)$.

Коментар. У много случајева једноставан поступак инспирисан обичним алгоритмом (оним којим се ручно решава проблем) није ефикасан за велике улазе. Понекад је пак такав приступ не само једноставан, него и ефикасан.

Приступ решавању проблема поступком који се визуелно намеће је обично добар избор. Тиме се може доћи до више корисних запажања о проблему. У овом случају, посматрајући слику, приметили смо да се проблем може решити пратећи неки пут од тачке до спољашњости многоугла.

Интересантан проблем јесте како наћи "оптималну" полуправу кроз дату тачку q, односно ону која сече најмањи број страница многоугла. За свако теме p_i израчунавамо угао који полуправа $q - p_i-$ заклапа са x осом и темена сортирамо растуће према овој вредности. За сваку страницу разликујемо почетно и крајње теме, при чему почетним теменом називамо оно теме коме одговара мањи угао полуправе. Најпре се одреди број b пресека полуправе кроз q и прво теме (у сортираном редоследу) са многоуглом (то се може урадити у времену $O(n)$, где је n број страница многоугла). Претпоставимо да је за i-то теме у сортираном редоследу, $i \geq 1$, одређен број пресека полуправе $q-p_i-$ са страницама многоугла. Приликом преласка на наредно теме p_{i+1} потребно је размотрити две странице кроз то теме и ажурирати вредност бројача b. Вредност бројача b се може притом променити за највише 2. У сваком случају сложеност ажурирања бројача је $O(1)$. Током проласка кроз сва темена памтимо најмању до сада добијену вредност бројача b и за који скуп полуправих се она добија (између која два темена треба да пролази дати зрак). С обзиром да је операција ажурирања константне временске сложености, а да то радимо за свако теме, односно $O(n)$ пута, укупна сложеност овог корака је $O(n)$. Укупна сложеност овог алгоритма је због почетног сортирања $O(n \log n)$.

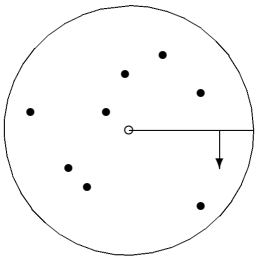
2.3 Конструкција простог многоугла

Скуп тачака у равни дефинише много различитих многоуглова, зависно од изабраног редоследа тачака. Размотрићемо сада проналажење простог многоугла са задатим скупом темена.

Проблем. Дато је n тачака у равни, таквих да нису све колинеарне. Повезати их простим многоуглом.

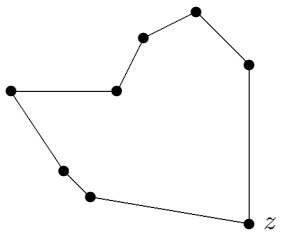
Постоји више метода за конструкцију траженог простог многоугла; уосталом, јасно је да у општем случају проблем нема једнозначно решење. Приказаћемо најпре геометријски приступ овом проблему. Нека је C неки круг, чија унутрашњост садржи све тачке. За налажење таквог круга довољно је $O(n)$ операција — израчунавања највећег међу растојањима произвољне тачке равни (центра круга) до свих n тачака. Површина C може се "пребрисати" (прегледати) ротирајућом полуправом којој је почетак центар C, видети слику 2.3. Претпоставимо за тренутак да ротирајућа полуправа у сваком тренутку садржи највише једну тачку. Очекујемо да ћемо спајањем тачака оним редом којим полуправа наилази на њих добити прост многоугао. Покушајмо да то докажемо. Означимо тачке, уређене у складу са редоследом наиласка полуправе на њих, са $p_1, p_2, \ldots, p_n$ (прва тачка бира се произвољно). За свако i, $1 \leq i \leq n$, страница p_i-p_{i-1} (односно $p_1 - p_n$ за $i = 1$) садржана је у новом (дисјунктном) исечку круга, па се не сече ни са једном другом страном. Ако би ово тврђење било тачно, добијени многоугао би морао да буде прост. Међутим, угао између полуправих кроз неке две узастопне тачке p_i и p_{i+1} може да буде већи од π. Тада исечак који

садржи дуж $p_i - p_{i+1}$ садржи више од пола круга и није конвексна фигура, а дуж $p_i - p_{i+1}$ пролази кроз друге исечке круга, па може да сече друге странице многоугла. Да бисмо се уверили да је то могуће, довољно је да уместо нацртаног замислимо круг са центром ван круга са слике 2.3. Ово је добар пример специјалних случајева на које се наилази при решавању геометријских проблема. Потребно је да будемо пажљиви, да бисмо били сигурни да су сви случајеви размотрени. На ову појаву наилази се код свих врста алгоритама, али је она код геометријских алгоритама драстичније изражена.



Слика 2.3: Пролазак тачака у кругу ротирајућом полуправом.

Да би се решио уочени проблем, могу се, на пример, фиксирати произвољне три тачке из скупа, а за центар круга изабрати нека тачка унутар њима одређеног троугла (на пример тежиште, које се лако налази). Овакав избор гарантује да ни један од добијених сектора круга неће имати угао већи од π. Могуће је изабрати и друго решење, да се за центар круга узме једна од тачака из скупа — тачка z са највећом x-координатом (и са најмањом y-координатом, ако има више тачака са највећом x-координатом); зваћемо је *екстремна тачка*. Затим користимо исти основни алгоритам. Сортирамо тачке према положају у кругу са центром z. Прецизније, сортирају се *углови* између x-осе и полуправих од z ка осталим тачкама. Ако две или више тачака имају исти угао, оне се даље сортирају према растојању од тачке z. На крају, z се повезује са тачком са најмањим и највећим углом, а остале тачке повезују се у складу са добијеним уређењем, по две узастопне. Пошто све тачке леже лево од z, до дегенерисаног случаја о коме је било речи не може доћи. Прост многоугао добијен овим поступком за тачке са слике 2.3 приказан је на слици 2.4.



Слика 2.4: Конструкција простог многоугла.

Описани метод може се усавршити на два начина. Угао φ који права $y = mx + b$ заклапа са x осом добија се коришћењем везе $m = \operatorname{tg}\varphi$, односно

из једначине $\varphi = \operatorname{arctg} m$. Међутим, углови се не морају експлицитно израчунавати. Углови се користе само за налажење редоследа којим треба повезати тачке. Исти редослед добија се уређењем *нагиба* (односно односа прираштаја y- и x-координата) одговарајућих полуправих; то чини непотребним израчунавање аркустангенса. Из истог разлога непотребно је израчунавање растојања кад две тачке имају исти нагиб — довољно је израчунати квадрате растојања. Дакле, нема потребе за израчунавањем квадратних коренова.

Алгоритам Prost_mnogougao$(p_1, p_2, \ldots, p_n)$;
Улаз: $p_1, p_2, \ldots, p_n$ (тачке у равни).
Излаз: P (прост многоугао са теменима $p_1, p_2, \ldots, p_n$ у неком редоследу).
begin
  променити ознаке тако да p_1 буде екстремна тачка;
  {тачка са највећом x-координатом, а ако таквих има више,}
  {она од њих која има најмању y-координату}
  for $i := 2$ **to** n **do**
    израчунати угао α_i између праве $-p_1 - p_i-$ и x-осе;
  сортирати тачке према угловима $\alpha_2, \ldots, \alpha_n$;
  {у групи са истим углом сортирати их према растојању од p_1}
  P је многоугао дефинисан сортираном листом тачака
end

Сложеност. Основна компонента временске сложености овог алгоритма потиче од сортирања. Сложеност алгоритма је дакле $O(n \log n)$.

2.4 Конвексни омотач

Конвексни омотач коначног скупа тачака дефинише се као најмањи конвексни многоугао који садржи све тачке скупа. Конвексни омотач се представља на исти начин као обичан многоугао, теменима наведеним у цикличком редоследу.

Проблем. Конструисати конвексни омотач задатих $n \geq 3$ тачака у равни.

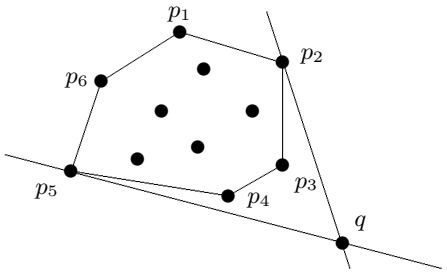
Обрада конвексних многоуглова једноставнија је од обраде произвољних многоуглова. На пример, постоји алгоритам сложености $O(\log n)$ за проверу припадности тачке конвексном n-тоуглу.

Претпоставимо да је задатак да утврдимо да ли се тачка q налази унутар датог конвексног многоугла $p_1 p_2 \ldots p_n$. Идеја на којој се заснива алгоритам је да се примени бинарна претрага: ако знамо да се q налази унутар угла $\angle p_i p_1 p_j$, $i < j$, онда се после провере да ли се q и p_j налазе са исте стране праве $p_1 p_m$, $m = \lfloor (i + j)/2 \rfloor$, зна да ли је q унутар угла $\angle p_m p_1 p_j$ (ако је одговор "да", или ако је q на правој $p_1 p_m$), или унутар угла $\angle p_i p_1 p_m$ (у противном). На почетку се проверава да ли је q унутар угла $\angle p_2 p_1 p_n$ (ако није, онда q не припада многоуглу). После највише $O(\log n)$ корака закључује се или да q не припада многоуглу, или да је унутар неког угла $\angle p_i p_1 p_{i+1}$; у другом случају је q у многоуглу акко припада троуглу $\triangle p_i p_1 p_{i+1}$.

Темена конвексног омотача су неке од тачака из задатог скупа. Кажемо да тачка *припада* омотачу ако је теме омотача. Ковексни омотач може се састојати од најмање три, а највише n тачака. Конвексни омотачи имају широку примену, па су због тога развијени многобројни алгоритми за њихову конструкцију.

2.4.1 Директни приступ

Као и обично, покушаћемо најпре са директним индуктивним приступом. Конвексни омотач за три тачке лако је наћи. Претпоставимо да умемо да конструишемо конвексни омотач скупа од $< n$ тачака, и покушајмо да конструишемо конвексни омотач скупа од n тачака. Како n-та тачка може да промени конвексни омотач за првих $n-1$ тачака? Постоје два могућа случаја: или је нова тачка у претходном конвексном омотачу (тада он остаје непромењен), или је она ван њега, па се омотач "шири" да обухвати и нову тачку, видети слику 2.5. Потребно је дакле решити два потпроблема: утврђивање да ли је нова тачка унутар омотача и проширивање омотача новом тачком. Они нису једноставни. Ствар се може упростити погодним избором n-те тачке. Звучи изазовно покушати са избором тачке унутар омотача; то, међутим, није увек могуће јер у неким случајевима све тачке припадају омотачу. Друга могућност, која се показала успешном у претходном проблему, је избор екстремне тачке за n-ту тачку.



Слика 2.5: Проширивање конвексног омотача новом тачком.

Изаберимо поново тачку са највећом x-координатом (и минималном y-координатом ако има више тачака са највећом x-координатом). Нека је то тачка q. Јасно је да тачка q мора бити теме конвексног омотача. Питање је како променити конвексни омотач осталих $n-1$ тачака тако да обухвати и q. Потребно је најпре пронаћи темена старог омотача која су у унутрашњости новог омотача (p_3 и p_4 на слици 2.5) и уклонити их; затим се умеће ново теме q између два постојећа (p_2 и p_5 на слици 2.5). **Права ослонца** конвексног многоугла је права која многоугао сече у само једном темену или у два суседна темена. Многоугао увек цео лежи са једне стране своје праве ослонца. Посматрајмо сада праве ослонца $-q-p_2-$ и $-q-p_5-$ на слици 2.5. Обично само два темена многоугла повезивањем са q одређују праве ослонца (игнорисаћемо специјални случај кад два темена многоугла леже на истој правој са q). Многоугао лежи између две праве ослонца, што указује како треба извести модификацију омотача. Праве ослонца заклапају минимални и максимални угао са x-осом међу свим правим

кроз q и неко теме многоугла. Да бисмо одредили та два темена, треба да размотримо праве из q ка свим осталим теменима, да израчунамо углове које оне заклапају са x-осом, и међу тим угловима изаберемо минималан и максималан. После проналажења два екстремна темена p_i, p_j лако је конструисати модификовани омотач. Потребно је елиминисати један од два добијена сегмента низа темена. Посматрају се суседна темена нпр. темена p_i. Једно од њих је са исте стране праве $-p_i - p_j-$ као и тачка q; то теме припада сегменту низа темена које треба заменити новим страницама $p_i - q$ и $q - p_j$. У примеру на слици 2.5 теме p_4, суседно екстремном темену p_5, налази се са исте стране дијагонале $p_5 - p_2$, па се сегмент старог омотача p_3, p_4 замењује са две нове странице $p_5 - q$ и $p_2 - q$.

Сложеност. За сваку тачку треба израчунати углове правих ка свим претходним теменима и x-осе, пронаћи минимални и максимални угао, додати нови чвор и избацити неке чворове. Сложеност додавања k-те тачке је дакле $O(k)$, а видели смо већ да је решење диференцне једначине $T(n) = T(n-1)+cn$ облика $O(n^2)$. Према томе, сложеност овог алгоритма је $O(n^2)$. Алгоритам на почетку такође захтева и сортирање, али је време сортирања асимптотски мање од времена потребног за остале операције.

2.4.2 Увијање поклона

Како се може побољшати описани алгоритам? Кад проширујемо многоугао теме по теме, доста времена трошимо на формирање конвексних многоуглова од тачака које могу бити унутрашње за коначни конвексни омотач. Може ли се то избећи? Уместо да правимо омотаче подскупова датог скупа тачака, можемо да посматрамо комплетан скуп тачака и да директно правимо његов конвексни омотач. Може се, као и у претходном алгоритму, кренути од екстремне тачке (која увек припада омотачу), пронаћи њој суседна темена омотача налазећи праве ослонца, и наставити на исти начин од тих суседа. Овај алгоритам из разумљивих разлога зове се **увијање поклона**.Полази се од једног темена "поклона", и онда се он увија у конвексни омотач проналазећи теме по теме омотача. Алгоритам описује наредни код. Алгоритам се може преправити тако да ради и у просторима веће димензије.

Алгоритам увијање поклона је директна последица примене следеће индуктивне хипотезе (по k):

Индуктивна хипотеза. За задати скуп од n тачака у равни, умемо да пронађемо конвексни пут дужине $k < n$ који је део конвексног омотача скупа.

Код ове хипотезе нагласак је на проширивању *пута*, а не омотача. Уместо да проналазимо конвексне омотаче мањих скупова, ми проналазимо део коначног конвексног омотача.

Сложеност. Да бисмо додали k-то теме омотачу, треба да пронађемо праву са најмањим углом у скупу од $n - k$ правих. Због тога је временска сложеност алгоритма увијање поклона $O(n^2)$, односно ако коначни омотач има m тачака, сложеност је $O(mn)$.

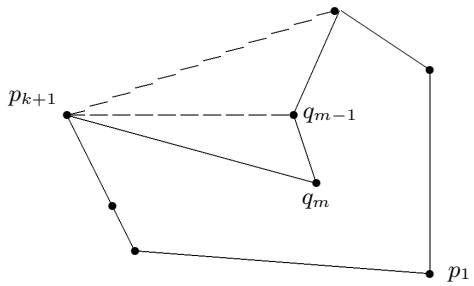
Алгоритам Uvijanje_poklona$(p_1, p_2, \ldots, p_n)$;
Улаз: $p_1, p_2, \ldots, p_n$ (скуп тачака у равни).
Излаз: P (конвексни омотач тачака $p_1, p_2, \ldots, p_n$).
begin
Нека је P празан скуп;
Нека је p тачка са највећом x-координатом
(и са најмањом y-координатом, ако има више
тачка са највећом x-координатом);
Укључи p у скуп P;
Нека је L права кроз p паралелна са x-осом;
while омотач P није завршен **do**
Нека је q тачка за коју је најмањи угао између L и $-p-q-$;
Укључи q у скуп P;
$L :=$ права $-p-q-$;
$p := q$
end

2.4.3 Грејемов алгоритам

Сада ћемо размотрити алгоритам за налажење конвексног омотача сложености $O(n \log n)$. Започиње се сортирањем тачака према угловима, слично као при конструкцији простог многоугла у одељку 2.3. Нека је p_1 тачка са највећом x-координатом (и са најмањом y-координатом, ако има више тачака са највећом x-координатом). За сваку тачку p_i израчунавамо угао између праве $-p_1 - p_i-$ и x-осе, и сортирамо тачке према величини ових углова, видети слику 2.6. Тачке пролазимо редоследом којим се појављују у (простом) многоуглу, и покушавамо да идентификујемо темена конвексног омотача. Као и код алгоритма увијање поклона памтимо пут састављен од дела прођених тачака. Прецизније, то је конвексни пут чији конвексни многоугао садржи све до сада прегледане тачке (одговарајући конвексни многоугао добија се повезивањем прве и последње тачке пута). Због тога, у тренутку кад су све тачке прегледане, конвексни омотач скупа тачака је конструисан. Основна разлика између овог алгоритма и увијања поклона је у чињеници да текући конвексни пут не мора да буде део коначног конвексног омотача. То је само део конвексног омотача до сада прегледаних тачака. Пут може да садржи тачке које не припадају коначном конвексном омотачу; те тачке биће елиминисане касније. На пример, пут од p_1 до q_m на слици 2.6 је конвексан, али q_m и q_{m-1} очигледно не припадају конвексном омотачу. Ово разматрање сугерише алгоритам заснован на следећој индуктивној хипотези.

Индуктивна хипотеза. Ако је дато n тачака у равни, уређених према алгоритму *Prost_mnogougao* (одељак 2.3), онда умемо да конструишемо конвексни пут преко неких од првих k тачака, такав да одговарајући конвексни многоугао обухвата првих k тачака.

Случај $k = 1$ је тривијалан. Означимо конвексни пут добијен (индуктивно) за првих k тачака са $P = q_1, q_2, \ldots, q_m$. Сада треба да проширимо индуктивну хипотезу на $k+1$ тачака. Посматрајмо угао између правих $-q_{m-1} - q_m-$ и



Слика 2.6: Грејемов алгоритам за налажење конвексног омотача скупа тачака.

$-q_m-p_{k+1}-$ (видети слику 2.6). Ако је тај угао мањи од π (при чему се угао мери из *унутрашњости* многоугла), онда се p_{k+1} може додати постојећем путу (нови пут је због тога такође конвексан), чиме је корак индукције завршен. У противном, тврдимо да q_m лежи у многоуглу добијеном заменом q_m у P тачком p_{k+1}, и повезивањем p_{k+1} са p_1. Ово је тачно јер су тачке уређене преме одговарајућим угловима. Права $-p_1 - p_{k+1}-$ лежи "лево" од првих k тачака. Због тога q_m јесте унутар горе дефинисаног многоугла, може се избацити из P, а p_{k+1} се може додати. Да ли је тиме све урађено? Не сасвим. Иако се q_m може елиминисати, модификовани пут не мора увек да буде конвексан. Заиста, слика 2.6 јасно показује да постоје случајеви кад треба елиминисати још тачака. На пример, тачка q_{m-1} може да буде унутар многоугла дефинисаног модификованим путем. Морамо да (уназад) наставимо са проверама последње две гране пута, све док угао између њих не постане мањи од π. Пут је тада конвексан, а хипотеза је проширена на $k+1$ тачку.

Алгоритам Grahamov_algoritam$(p_1, p_2, \ldots, p_n)$;
Улаз: $p_1, p_2, \ldots, p_n$ (скуп тачака у равни).
Излаз: $q_1, q_2, \ldots, q_m$ (конвексни омотач тачака $p_1, p_2, \ldots, p_n$).
begin
  Нека је p_1 тачка са највећом x-координатом (а најмањом y-координатом,
    ако има више тачака са највећом x-координатом);
  Помоћу алгоритма *Prost_mnogougao* уредити тачке у односу на p_1;
    нека је то редослед $p_1, p_2, \ldots, p_n$;
  $q_1 := p_1$;
  $q_2 := p_2$;
  $q_3 := p_3$; {пут P се на почетку састоји од p_1, p_2 и p_3}
  $m := 3$;
  for $k := 4$ **to** n **do**
    while угао између $-q_{m-1}-q_m-$ и $-q_m-p_k-) \geq \pi$ **do**
      $m := m - 1$;
    $m := m + 1$;
    $q_m := p_k$ {на слици 2.6 то је p_{k+1}}
end

Сложеност. Главни део сложености алгоритма потиче од почетног сор-

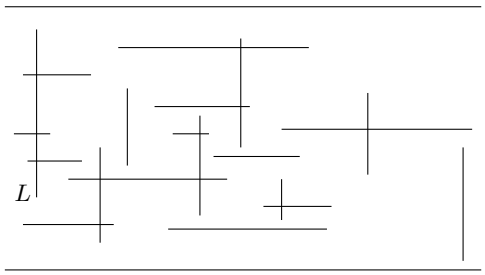
тирања. Остатак алгоритма извршава се за време $O(n)$. Свака тачка скупа разматра се тачно једном у индуктивном кораку као p_{k+1}. У том тренутку тачка се увек додаје конвексном путу. Иста тачка биће разматрана и касније (можда чак и више него једном) да би се проверила њена припадност конвексном путу. Број тачака на које се примењује овакав повратни тест може бити велики, али се све оне, сем две (текућа тачка и тачка за коју се испоставља да даље припада конвексном путу) елиминишу, јер тачка може бити елиминисана само једном! Према томе, троши се највише константно време за елиминацију сваке тачке и константно време за њено додавање. Укупно је за ову фазу потребно $O(n)$ корака. Због сортирања је време извршавања комплетног алгоритма $O(n \log n)$.

2.5 Пресеци хоризонталних и вертикалних дужи

Често се наилази на проблеме налажења пресека. Понекад је потребно проналажење пресека више објеката, а понекад само треба открити да ли је пресек непразан скуп. Проблеми детекције су обично лакши. У овом одељку приказаћемо један проблем налажења пресека, који илуструје важну технику за решавање геометријских проблема. Иста техника може се применити и на друге проблеме.

Проблем. За задати скуп од n хоризонталних и m вертикалних дужи пронаћи све њихове пресеке.

Овај проблем важан је, на пример, при пројектовању кола VLSI (интегрисаних кола са огромним бројем елемената). Коло може да садржи на хиљаде "жичица", а пројектант треба да буде сигуран да не постоје неочекивани пресеци. На проблем се такође наилази при елиминацији скривених линија када је потребно закључити које ивице или делови ивица су скривени самим објектом или неким другим објектом; тај проблем је обично компликованији, јер се не ради само о хоризонталним и вертикалним линијама. Пример улаза за овај проблем приказан је на слици 2.7.



Слика 2.7: Пресеци хоризонталних и вертикалних дужи.

Налажење свих пресека вертикалних дужи је једноставан проблем, који се оставља читаоцу за вежбање (исто се односи и на хоризонталне дужи). Претпоставимо због једноставности да не постоје пресеци између произвољне две хоризонталне, односно произвољне две вертикалне дужи. Ако покушамо да проблем решимо уклањањем једне по једне дужи (било хоризонталне, било вертикалне), онда ће бити неопходно да се пронађу пресеци уклоњене

дужи са свим осталим дужима, па се добија алгоритам са $O(mn)$ налажења пресека дужи. У општем случају број пресека може да буде $O(mn)$, па алгоритам може да утроши време $O(mn)$ већ само за приказивање свих пресека. Међутим, број пресека може да буде много мањи од mn. Волели бисмо да конструишемо алгоритам који ради добро кад има мало пресека, а не превише лоше ако пресека има много. Дакле циљ нам је да проблем решимо коришћењем алгоритма осетљивог на излаз, чија сложеност зависи и од величине улаза и од величине излаза. То се може постићи комбиновањем двеју идеја: избора специјалног редоследа индукције и појачавања индуктивне хипотезе.

Редослед примене индукције може се одредити покретном правом (енг. sweeping line) која прелази (”скенира”) раван слева удесно; дужи се разматрају оним редом којим на њих наилази покретна права. Поред налажења пресечних тачака, треба чувати и неке податке о дужима које је покретна права већ захватила. Ти подаци биће корисни за ефикасније налажење наредних пресека. Ова техника зове се **техника покретне линије**.

Замислимо вертикалну праву која прелази раван слева удесно. Да бисмо остварили ефекат пребрисавања, сортирамо све крајеве дужи према њиховим x-координатама. Две крајње тачке вертикалне дужи имају исте x-координате, па се региструје само једна x-координата За хоризонталне дужи морају се користити оба краја. После сортирања крајева, дужи се разматрају једна по једна утврђеним редоследом. Као и обично при индуктивном приступу, претпостављамо да смо пронашли пресечне тачке претходних дужи, и да смо обезбедили неке допунске информације, па сада покушавамо да обрадимо следећу дуж и да унесемо неопходне допуне информација. Према томе структура алгоритма је у основи следећа. Разматрамо крајеве дужи један по један, слева удесно. Користимо информације прикупљене до сада (нисмо их још специфицирали) да обрадимо крај дужи, пронађемо пресеке у којима она учествује, и допуњујемо информације да бисмо их користили при следећем наиласку на неки крај дужи. Основни проблем је дефинисање информација које треба прикупљати. Покушајмо да покренемо алгоритам да бисмо открили које су то информације потребне.

Природно је у индуктивну хипотезу укључити познавање свих пресечних тачака дужи које се налазе лево од тренутног положаја покретне праве. Да ли је боље проверавати пресеке кад се разматра хоризонтална или вертикална дуж? Кад разматрамо вертикалну дуж, хоризонталне дужи које је могу сећи још увек се разматрају (пошто није достигнут њихов десни крај). С друге стране, кад посматрамо било леви, било десни крај хоризонталне дужи, ми или нисмо наишли на вертикалне дужи које је секу, или смо их већ заборавили. Дакле, боље је пресеке бројати приликом наиласка на вертикалну дуж. Претпоставимо да је покретна права тренутно преклопила вертикалну дуж L (видети слику 2.7). Какве информације су потребне да се пронађу сви пресеци дужи L? Пошто се претпоставља да су сви пресеци лево од тренутног положаја покретне праве већ познати, нема потребе разматрати хоризонталну дуж ако је њен десни крај лево од покретне праве. Према томе, треба разматрати само оне хоризонталне дужи чији су леви крајеви лево, а десни крајеви десно од тренутног положаја покретне праве (на слици 2.7 таквих дужи има шест). Потребно је чувати листу таквих хоризонталних дужи (односно њихових y-координата). Кад се наиђе на вертикалну дуж L, потребно је проверити да ли се она сече са

тим хоризонталним дужима. Важно је приметити да за налажење пресека са L овде x-координате крајева дужи нису од значаја. Ми већ знамо да хоризонталне дужи из листе имају x-координате које "покривају" x-координату дужи L. Потребно је проверити само y-координате хоризонталних дужи из листе, да би се проверило да ли су оне обухваћене опсегом y-координата дужи L. Сада смо спремни да формулишемо индуктивну хипотезу.

Индуктивна хипотеза. Нека је задата листа од првих k сортираних x-координата крајева дужи као што је описано, при чему је x_k највећа од x-координата. Умемо да пронађемо све пресеке дужи који су лево од x_k, и да елиминишемо све хоризонталне дужи које су лево од x_k.

За хоризонталне дужи које се још увек разматрају рећи ћемо да су **кандидати** (то су хоризонталне дужи чији су леви крајеви лево, а десни крајеви десно од текућег положаја покретне праве). Формираћемо и одржавати структуру података која садржи скуп кандидата. Одложићемо за тренутак анализу реализације ове структуре података.

Базни случај за наведену индуктивну хипотезу је једноставан. Да бисмо је проширили, потребно је да обрадимо $(k+1)$-и крај дужи. Постоје три могућности.

1. $(k+1)$-и крај дужи је десни крај хоризонталне дужи; дуж се тада просто елиминише из списка кандидата. Као што је речено, пресеци се проналазе при разматрању вертикалних дужи, па се ни један од пресека не губи елиминацијом хоризонталне дужи. Овај корак дакле проширује индуктивну хипотезу.

2. $(k+1)$-и крај дужи је леви крај хоризонталне дужи; дуж се тада додаје у списак кандидата. Пошто десни крај дужи није достигнут, дуж се не сме још елиминисати, па је и у овом случају индуктивна хипотеза проширена на исправан начин.

3. $(k+1)$-и крај дужи је вертикална дуж. Ово је основни део алгоритма. Пресеци са овом вертикалном дужи могу се пронаћи упоређивањем y-координата свих хоризонталних дужи из скупа кандидата са y-координатама крајева вертикалне дужи.

Алгоритам је сада комплетан. Број упоређивања ће обично бити много мањи од mn. На жалост, у најгорем случају овај алгоритам ипак захтева $O(mn)$ упоређивања, чак иако је број пресека мали. Ако се, на пример, све хоризонталне дужи простиру слева удесно ("целом ширином"), онда се мора проверити пресек сваке вертикалне дужи са свим хоризонталним дужима, што имплицира сложеност $O(mn)$. Овај најгори случај појављује се чак и ако ниједна вертикална дуж не сече ниједну хоризонталну дуж.

Да би се усавршио алгоритам, потребно је смањити број упоређивања y-координата вертикалне дужи са y-координатама хоризонталних дужи у скупу кандидата. Нека су y-координате дужи која се тренутно разматра y_D и y_G, и нека су y-координате хоризонталних дужи из скупа кандидата $y_1, y_2, \ldots, y_r$. Претпоставимо да су хоризонталне дужи у скупу кандидата задате сортирано према y-координатама (односно низ $y_1, y_2, \ldots, y_r$ је растући). Хоризонталне дужи које се секу са вертикалном могу се пронаћи извођењем две бинарне претраге, једне за y_D, а друге за y_G. Нека је $y_i < y_D \leq y_{i+1} \leq$

$y_j \leq y_G < y_{j+1}$. Вертикалну дуж секу хоризонталне са координатама y_{i+1}, y_{i+2},... y_j, и само оне. Може се такође извршити једна бинарна претрага за нпр. y_D, а затим пролазити y-координате док се не дође до y_j. Иако је полазни проблем дводимензионалан, налажење y_{i+1}, ... y_j је једнодимензионални проблем. Тражење бројева у једнодимензионалном опсегу (у овом случају од y_D до y_G) зове се *једнодимензионална претрага опсега*. Ако су бројеви сортирани, онда је време извршења једнодимензионалне претраге опсега пропорционално збиру времена тражења првог пресека и броја пронађених пресека. Наравно, не можемо да приуштимо себи сортирање хоризонталних дужи при сваком наиласку на вертикалну дуж.

Присетимо се још једном захтева. Потребна је структура података погодна за чување кандидата, која дозвољава уметање новог елемента, брисање елемента и ефикасно извршење једнодимензионалне претраге опсега. На срећу, постоји више структура података — на пример, уравнотежено стабло — која омогућују извршење уметања, брисања и тражења сложености $O(\log n)$ по операцији (где је n број елемената у скупу), и линеарно претраживање за време пропорционално броју пронађених елемената. Претрага започиње тражењем y_D и y_G у стаблу које садржи кандидате $y_1, \ldots, y_r$. Нека су путеви који се при томе прелазе полазећи од корена стабла означени са P_D, P_G, и нека је v последњи заједнички чвор за та два пута. Резултат претраге су кључеви комплетних десних подстабала чворова са пута P_D (испод v) у којима пут наставља левом граном, и (симетрично) комплетних левих подстабала чворова са пута P_G (испод v) у којима пут наставља десном граном. Овим су обухваћени сви чворови у унутрашњим подстаблима која су ограђена путевима P_D и P_G. Током обиласка, такође се проверавају и пријављују сви чворови на путевима P_D и P_G који леже у интервалу.

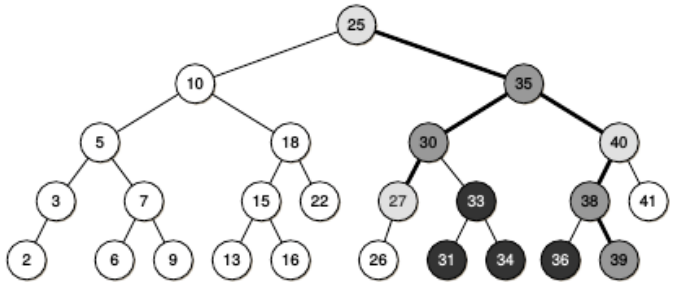
Алгоритам BSP_pretraga_opsega(v, x', x'')
Улаз. v (показивач на корен БСП), $[x', x'']$ (интервал)
Излаз. F (скуп тачака из v које припадају интервалу)

```
1  if v ≠ nil:
2      if x' ≤ v.kljuc:
3          BSP_pretraga_opsega(v.levi, x', x'')
4      if x' ≤ v.kljuc and x'' ≥ v.kljuc:
5          додај v.kljuc у F
6      if x'' > v.kljuc:
7          BSP_pretraga_opsega(v.desni, x', x'')
```

Сложеност. Почетно сортирање према x-координатама крајева дужи захтева $O((m+n)\log(m+n))$ корака. Пошто свако уметање и брисање захтева $O(\log n)$ корака, укупно време за обраду хоризонталних дужи је $O(n \log n)$. Обрада вертикалних дужи захтева једнодимензионалну претрагу опсега, која се може извршити за време $O(\log n + r)$, где је r број пресека ове вертикалне дужи. Временска сложеност алгоритма је дакле

$$O((m+n)\log(m+n) + R),$$

где је R укупан број пресека.



Слика 2.8: Пример једнодимензионалне претраге опсега. Кандидати, њих 25, уписани су у уравнотежено стабло, а исписују се кандидати из интервала [30,39]. Резултат чине тамно сиви и црни чворови.

Алгоритам Preseci$((v_1, v_2, \ldots, v_m), (h_1, h_2, \ldots, h_n))$;
Улаз: $v_1, v_2, \ldots, v_m$ (скуп вертикалних дужи), и$h_1, h_2, \ldots, h_n$ (скуп хоризонталних дужи).
$\{y_G(v_i)$ и $y_G(v_i)$ су мања и већа y-координата дужи $v_i\}$
Излаз: Скуп свих парова дужи које имају пресек.
begin
Сортирати све x-координате у неопадајући низ и сместити их у Q;
$V := \emptyset$;
{V је скуп хоризонталних дужи, тренутних кандидата за пресеке; организован}
{је као уравнотежено стабло према y-координатама хоризонталних дужи}
while Q је непразан **do**
уклонити из Q наредни крај дужи, p;
if p је десни крај дужи h_k **then**
уклонити h_k из V
else if p је леви крај дужи h_k **then**
укључити h_k у V
else if p је x-координата вертикалне дужи v_i **then**
{налажење пресека кандидата са вертикалном дужи}
једнодимензионална претрага опсега од $y_G(v_i)$ до $y_G(v_i)$ у V
end

2.6 Одређивање две најудаљеније тачке у равни

Често је, поред две најближе тачке, важно одредити и две најудаљеније тачке у неком скупу тачака. Нека је S скуп од n тачака у равни и нека је задатак одредити дијаметар $D(S)$ скупа тачака S који се дефинише као највеће растојање неке две тачке из S:

$$D(S) = \max\{d(p,q) \mid p,q \in S\}$$

Притом желимо да утврдимо и које су то две тачке a и b за које важи да је $d(a,b) = D(S)$ (можемо да разликујемо две варијанте овог проблема: наћи било који пар таквих тачака или све парове који задовољавају овај услов). Директан начин да решимо овај проблем састојао би се од рачунања растојања $d(p,q)$ за сваки пар тачака (p,q) и одређивања највеће од тих вредности. С обзиром на то да се рачунање растојања две тачке може урадити у константном времену и да је број парова тачака једнак $\binom{n}{2}$, овај алгоритам је сложености $O(n^2)$. Показаћемо у наставку текста да се овај проблем може решити и алгоритмом сложености $O(n \log n)$.

Лема 2.1. Дијаметар скупа тачака S једнак је дијаметру скупа темена његовог конвексног омотача S'.

Доказ. Нека су p и q две различите тачке скупа S тако да нису обе у S'. Без смањења општости претпоставимо да $q \notin S'$. Покажимо да онда важи да постоји тачка r у S' тако да је $d(p,q) < d(p,r)$. Без нарушавања општости, претпоставимо да је права кроз тачке p и q хоризонтална и да је p лево од q. С обзиром на то да тачка q не припада конвексном омотачу, мора да постоји теме конвексног омотача r које је на вертикалној правој кроз q или десно од ње. Неједнакост $d(p,q) < d(p,r)$ следи из чињеница да је у троуглу (p,q,r) угао код темена q прав или туп (а наспрам највећег угла у троуглу увек лежи највећа страница).

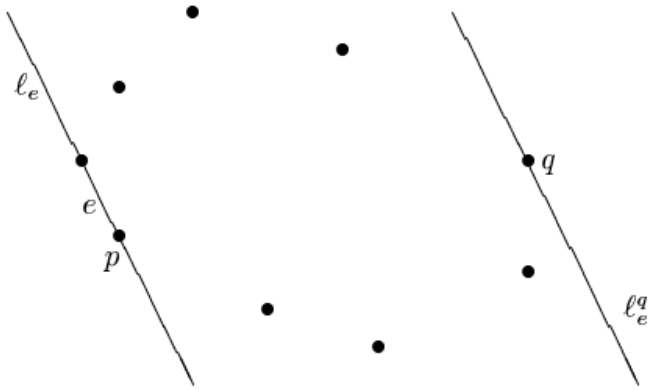
Дакле, да бисмо израчунали дијаметар скупа тачака у равни, довољно је размотрити само темена која припадају његовом конвексном омотачу. Другим речима, без смањења општости може се претпоставити да су тачке из скупа S темена конвексног многоугла, при чему су сви углови конвексног омотача мањи од π (ако је угао многоугла у неком темену једнак π, онда то теме не припада конвексном омотачу).

Лема 2.2. Нека је l права кроз тачке p_1 и p_2. Низ растојања преосталих тачака до праве l: $d(p_3,l), d(p_4,l), \ldots, d(p_n,l)$ расте до неке вредности $d(p_i,l)$, а затим опада.

Доказ. Ово тврђење следи директно из чињенице да су тачке скупа S темена конвексног многоугла.

Ако је e нека ивица конвексног омотача, означимо са l_e праву која садржи e. Нека је L скуп свих парова (p,q) за које за једну од ивица e конвексног омотача које садрже p важи да је q најдаља тачка конвексног омотача од l_e, односно важи да је: $d(q,l_e) = \max\{d(r,l_e) | r \in S\}$. У примеру на слици 2.9 тачка q је на максималном растојању од праве l_e. Ако са l_e^q означимо праву кроз q која је паралелна са l_e, тада су све тачке из S

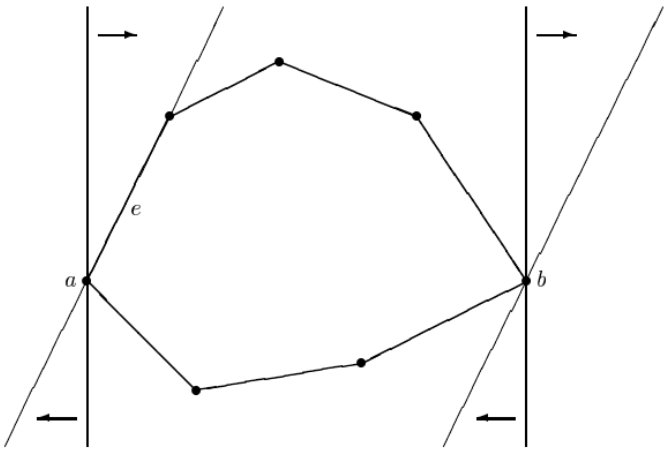
између l_e и l_e^q (због конвексности, права l_e^q може да садржи највише још једну тачку скупа S). Стога је пар (p, q) садржан у скупу L.



Слика 2.9: Илустрација дефиниције скупа L. Ивица e има p као крајњу тачку и q је најудаљенија од l_e. Стога скуп L садржи пар (p, q)

Лема 2.3. Дијаметар скупа S једнак је највећем растојању $d(p, q)$ за неки пар из скупа L.

Доказ. Нека су a и b две тачке из S такве да је $d(a, b) = D(S)$. Покажимо да је пар (a, b) или пар (b, a) садржан у L, чиме ће лема бити доказана.



Слика 2.10: Илустрација доказа леме 3. Права кроз a и e је прва која наилази на ивицу конвексног омотача.

Без нарушавања општости, претпоставимо да је права кроз a и b хоризонтална. Све тачке из $S \setminus \{a, b\}$ су строго између две вертикалне праве кроз a и b (иначе би дијаметар скупа S био већи од $d(a, b)$). Ротирајмо

ове две праве симултано у смеру казаље на часовнику око a и b и станимо чим се једна од њих поклопи са страницом конвексног омотача скупа S; приликом ротације ове две праве остају паралелне (слика 2.10). Нека је e ивица са којом се дешава прво преклапање. Ако је a крајња тачка ове ивице, није тешко закључити да b има највеће растојање до праве кроз e. Стога је пар (a, b) садржан у скупу L. Слично, уколико је b крајња тачка ивице e, онда је пар (b, a) садржан у L.

На основу претходне леме закључујемо да дијаметар можемо израчунати на следећи начин: најпре одредимо скуп L, а затим одредимо и сва растојања одређена елементима из L и пронађемо највеће. Покажимо да скуп L садржи највише $4n$ елемената, при чему је са n означен број ивица конвексног многоугла S. Уочимо једну ивицу e конвексног омотача. Она је ограничена двема тачкама. Постоје највише две тачке скупа S на максималном растојању од праве l_e кроз e. Последица ове чињенице је да овој ивици одговара највише четири пара у L из чега следи да је величина скупа L највише $4n$. На основу тога закључујемо да је други корак алгоритма сложености $O(n)$. Остаје проблем одређивања скупа L. Показаћемо да се он може израчунати у времену $O(n)$.

Подсетимо се да смо тачке скупа S нумерисали као $p_1, p_2, \ldots, p_n$ у смеру супротном од кретања казаљке на часовнику. Нека је e ивица ограничена тачкама p_1 и p_2 и нека је l_e права која садржи e. Желимо да пронађемо највише две тачке скупа S које су на максималном растојању од l_e. Лема 2.2 нам даје једноставан поступак за одређивање ових тачака:

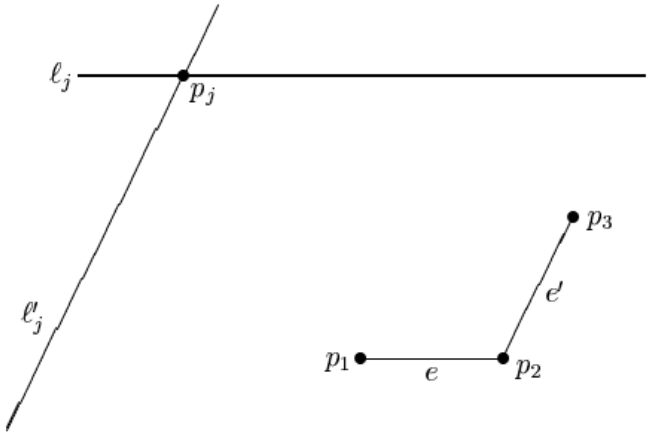
$j \leftarrow 3$
while $d(p_{j+1}, l_e) > d(p_j, l_e)$ **do**
  $j \leftarrow j + 1$

Притом се подразумева да се индекси тачака рачунају по модулу n, тј. да важи $p_{n+1} = p_1$. На крају ове while петље имамо да важи $d(p_3, l_e) < d(p_4, l_e) < \ldots < d(p_j, l_e)$ и $d(p_{j+1}, l_e) \leq d(p_j, l_e)$. Ако је $d(p_{j+1}, l_e) < d(p_j, l_e)$ онда је p_j једина тачка са максималним растојањем до l_e и додајемо два пара (p_1, p_j) и (p_2, p_j) у L. Иначе, имамо $d(p_{j+1}, l_e) = d(p_j, l_e)$ и додајемо четири пара (p_1, p_j), $(p_1, p_{j+1}), (p_2, p_j), (p_2, p_{j+1})$ у L. На овај начин смо обрадили прву ивицу e. Укупно утрошено време је $O(n)$, јер индекс j пролази n вредности.

Нека је e' наредна ивица конвексног омотача, која има крајње тачке p_2 и p_3. Размотримо како можемо да ефикасно пронађемо највише две тачке скупа S која су на максималном растојању од $l_{e'}$.

Лема 2.4. Нека је индекс j такав да важи да је p_j на највећем растојању од праве l_e. Слично, нека је индекс i такав да је тачка p_i на највећем растојању од l'_e. Тада је $j \leq i \leq n$ или $i = 1$.

Доказ. Претпоставимо без нарушавања општости да је права кроз p_1 и p_2 хоризонтална. Тада је p_1 лево од p_2. Нека је l_j (аналогно l'_j) права кроз p_j паралелна са l_e (односно l'_e), видети слику 2.11. С обзиром на то да је p_j на највећем растојању од l_e, знамо да су тачке $p_4, p_5, \ldots, p_{j-1}$ све на правој l_j или испод ње. Због конвексности ове тачке морају да буду десно од праве l'_j (показати да је ово тачно и ако је угао у p_2 мањи од $\pi/2$). Стога важи да је $d(p_j, l'_e) > d(p_k, l'_e), k = 4, 5, \ldots, j-1$, дакле $i \notin \{4, 5, \ldots, j-1\}$. Јасно је



Слика 2.11: Илустрација доказа леме 4.

и да је $i \neq 2$ и $i \neq 3$. Према томе, на најдаљу тачку од e' наилази се после тачке p_j.

Последица ове леме је следећа: да бисмо нашли индекс i, не морамо да кренемо од p_4 током обиласка конвексног омотача; можемо кренути од p_j.

У општем случају, нека је e_k ивица конвексног омотача са крајњим тачкама p_k и p_{k+1} и нека је p_u тачка на максималном растојању од праве l_k кроз e_k. Да бисмо пронашли тачку на максималном растојању од праве l_{k+1} која садржи наредну ивицу e_{k+1}, можемо размотрити темена конвексног омотача почев од p_u и размотрити тачке $p_u, p_{u+1}, \ldots, p_n, p_1, \ldots, p_k$ све док не пронађемо максимални елемент низа

$$d(p_u, l_{k+1}), d(p_{u+1}, l_{k+1}), \ldots, d(p_n, l_{k+1}), d(p_1, l_{k+1}), \ldots, d(p_k, l_{k+1})$$

за који знамо да му елементи прво расту, а затим опадају. На овај начин рачунамо скуп L и притом пролазимо максимално два пута око конвексног омотача, те се овај скуп рачуна у времену $O(n)$. Стога смо доказали наредну теорему:

Теорема 1. *Нека је S скуп од n тачака у равни. Дијаметар скупа S се може израчунати у времену $O(n \log n)$. Ако су тачке скупа S дате као темена конвексног многоугла, сортираним редоследом, онда се дијаметар може израчунати у времену $O(n)$.*

Коментар 2. Може постојати већи број тачака које су на максималном растојању, али сви парови таквих тачака морају бити садржани у скупу L. Стога може постојати највише $4n$ парова тачака које су на растојању $D(S)$ и предложени алгоритам их све проналази.

Размотримо још неколико занимљивих геометријских проблема.

2.7 Дужина уније дужи на правој

Проблем. За дати скуп од n дужи на правој, одредити укупну дужину коју покривају ове дужи.

Уколико би све дужи биле међусобно дисјунктне, онда бисмо проблем лако решили: сабрали бисмо дужине свих дужи. Међутим, ове дужи се могу преклапати (потенцијално и по неколико њих истовремено), те је потребно развити другачији алгоритам. Претпоставимо да све дужи леже на x-оси. Идеја је искористити технику вертикалне покретне праве која раван прелази слева удесно и која редом пролази скупом крајњих тачака свих датих дужи.

Формирајмо од свих x координата крајњих тачака дужи низ *tacke* дужине $2n$. Уз x координату тачке потребно је уз сваки члан низа памтити и информацију да ли је тачка почетна и крајња. Сортирајмо затим овај низ према x координати неопадајуће; уколико две или више тачака имају исту вредност x координате, прво се у сортираном низу требају налазити крајње тачке, а затим почетне. Биће нам потребан један наменски бројач br чија је вредност у сваком тренутку број преклопљених дужи непосредно иза покретне праве (а пре наредног краја дужи) и који на почетку иницијализујемо на 0, и променљива *ukupna_duzina* коју такође иницијализујемо на нулу и чија вредност треба да буде збир дужина дужи које је покретна права већ "прегледала". Ова два тврђења нам представљају инваријанту петље алгоритма који треба формулисати. Након тога пролазимо редом кроз елементе низа и за сваки члан низа уколико је вредност бројача већа од нуле променљиву *ukupna_duzina* увећавамо за вредност $tacka[i] - tacka[i-1]$. Приликом наиласка на почетну тачку дужи вредност бројача инкрементирамо, а приликом наиласка на крајњу тачку дужи вредност бројача декрементирамо (дакле у сваком тренутку вредност бројача нам указује на број дужи које се у тој тачки преклапају).

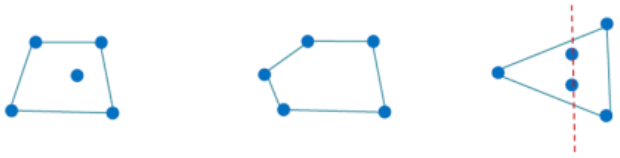
Овај алгоритам (Клее, 1977.) представља илустрацију технике покретне праве. Сложеност овог алгоритма је $O(n \log n)$ што потиче од почетног сортирања крајњих тачака дужи.

2.8 "Happy-ending" teorema

Теорема 3. *За сваки скуп од 5 тачака у равни које су у општем положају постоји четворочлани подскуп који формира конвексни четвороугао.*

Ова теорема се може једноставно доказати разматрањем случајева. Уколико су од ових пет тачака четири или пет тачака темена конвексног омотача, онда било које четири тачке конвексног омотача одређују решење. Ако скуп тачака има облик троугла са две тачке унутар њега, онда се увек могу одабрати две унутрашње тачке и једна од страница троугла. Коју страницу троугла бирамо? С обзиром на то да су тачке у општем положају (тј. никоје три тачке нису колинеарне), права која садржи две унутрашње тачке скупа сече тачно две странице троугла. Бирамо трећу страницу троугла и две унутрашње тачке.

На слици 2.12 приказане су све три могуће конфигурације на које можемо наићи приликом анализе случајева. Овај резултат је Пол Ердеш назвао



Слика 2.12: Различите конфигурације проблема: када се конвексни омотач састоји од пет, четири и три тачке

"happy-ending" теорема јер се двоје његових пријатеља који су радили на овом проблему (Џорџ Секереш и Естер Клајн) венчало. Уопштење овог резултата је теорема Ердеш-Секереш која гласи:

Теорема 4. *Најмањи број тачака за које сваки произвољан распоред садржи конвексни подскуп од n тачака је $2^{n-2}+1$.*

Она је и даље без доказа, али су доказане неке мање прецизне границе.

2.9 Испитивање да ли у скупу од n тачака постоје три колинеарне тачке

Проблем. Нека је задато n тачака у равни. Потребно је испитати да ли међу њима постоје неке три које су колинеарне.

Алгоритам грубе силе би за сваке три тачке проверавао колинеарност, па би сложеност била $O(n^3)$.

За сваку тачку израчунамо нагиб праве кроз њу и сваку од преосталих $n-1$ тачака скупа, а затим тај низ нагиба сортирамо. Након тога потребно је проверити да ли постоје две суседне исте вредности у овом сортираном низу. Ако постоје, онда постоје три колинеарне тачке. Сложеност овог алгоритма је $O(n)$ за рачунање нагиба правих, а затим $O(n \log n)$ за њихово сортирање. То радимо n пута, тако да је укупна сложеност алгоритма $O(n^2 \log n)$.

2.10 Испитивање да ли у скупу од n тачака постоји квадрат

Проблем. За датих n тачака у равни потребно је утврдити да ли постоје неке четири које образују квадрат.

Алгоритам грубе силе би за сваке четири тачке проверавао да ли образују квадрат и био би сложености $O(n^4)$. Алгоритам можемо унапредити тако што за сваке две тачке можемо да израчунамо координате потенцијалних квадрата над том страницом (постоје по два таква квадрата). Након тога, потребно је установити да ли се у скупу налазе тачке са датим координатама. Низ тачака можемо сортирати растуће и онда применити бинарну претрагу. Алгоритам је сложености $O(n^2 \log n)$.

2.11 Резиме

Геометријски алгоритми су на неки начин мање апстрактни од графовских, јер смо навикли да видимо и радимо са геометријским објектима. Међутим, то је само први утисак. Радити са огромним бројем објеката није исто што и радити са малим сликама, па морамо бити опрезни да нас представа коју имамо не наведе на погрешне закључке. Морају се предвидети многи специјални случајеви. Алгоритам за утврђивање припадности тачке унутрашњости многоугла (одељак 2.2) је добар пример. Због тога конструкција геометријских алгоритама захтева посебан опрез.

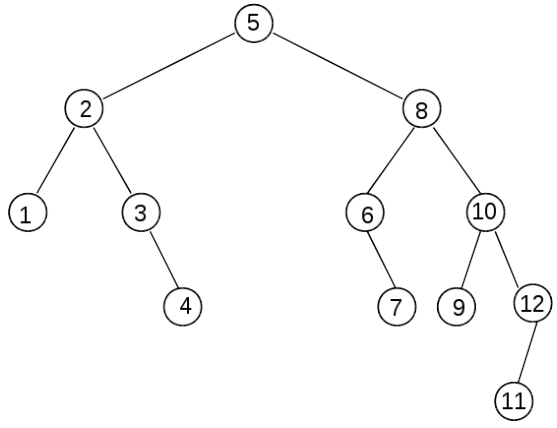
У конструкцији (дискретних) геометријских алгоритама важну улогу игра индукција. Техника покретне праве, заснована на индукцији, заједничка је за више геометријских алгоритама; такође се често користи разлагање. Геометријски алгоритми (изузев најједноставнијих) често захтевају компликоване структуре података, па су многе сложене структуре података развијене управо за ту намену. Овде нисмо разматрали ни једну од таквих специјалних структура података.

Глава 3

Структуре података

3.1 АВЛ стабла

АВЛ стабла (која су име добила по ауторима, Адельсон–Вельский и Ландис, 1962.) су структура података која гарантује да сложеност ни једне од операција тражења, уметања и брисања у **најгорем случају** није већа од $O(\log n)$, где је n број елемената. Идеја је уложити допунски напор да се после сваке операције стабло **уравнотежи**, тако да висина стабла увек буде $O(\log n)$. При томе се уравнотеженост стабла дефинише тако да се може лако одржавати. Прецизније, АВЛ стабло се дефинише као бинарно стабло претраге код кога је за сваки чвор апсолутна вредност разлике висина левог и десног подстабла мања или једнака од један (видети пример на слици 3.1).



Слика 3.1: Пример АВЛ стабла.

Као што показује следећа теорема, висина АВЛ стабла је $O(\log n)$.

Теорема 5. *За АВЛ стабло са n чворова висина h задовољава услов $h < 2\log_2 n$.*

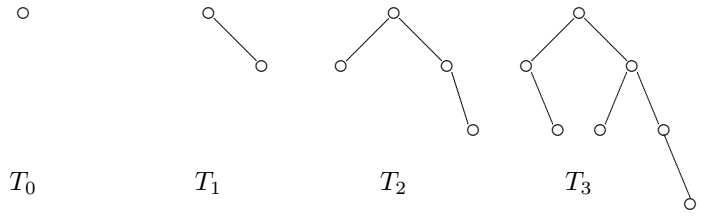
Доказ. Нека је T_h АВЛ стабло висине $h \geq 0$ са најмањим могућим бројем чворова. Стабло T_0 садржи само корен, а стабло T_1 корен и једног сина, рецимо десног. За $h \geq 2$ стабло T_h може се формирати на следећи начин: његово подстабло мање висине $h-2$ такође треба да буде АВЛ стабло са минималним бројем чворова, дакле T_{h-2}; слично, његово друго подстабло треба да буде T_{h-1}. Стабла описана оваквом "диференцном једначином" зову се *Фибоначијева стабла*. Неколико првих Фибоначијевих стабала, таквих да је у сваком унутрашњем чвору висина левог подстабла мања, приказано је на слици 3.2. Означимо са n_h минимални број чворова АВЛ стабла висине h, тј. број чворова стабла T_h; $n_0 = 1$, $n_1 = 2$, $n_2 = 4, \ldots$ По дефиницији Фибоначијевих стабала је $n_h = n_{h-1} + n_{h-2} + 1$, за $h \geq 2$. С обзиром на то да је $n_h > n_{h-1}$ за свако $h \geq 2$ важи:

$$n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} + 1 > 2n_{h-2}$$

Докажимо индукцијом да важи тврђење $n_h > 2^{h/2}$. За $h = 0$ и $h = 1$ тврђење важи. Претпоставимо да тврђење важи за $h-2$, односно да важи $n_{h-2} > 2^{(h-2)/2}$. На основу везе $n_h > 2n_{h-2}$ важи

$$n_h > 2 \cdot 2^{(h-2)/2} = 2^{h/2-1+1} = 2^{h/2}$$

односно након логаритмовања обе стране добијамо $h < 2\log_2 n_h \leq 2\log_2 n$ јер је по претпоставци за сва стабла висине h број чворова n већи или једнак од n_h.



Слика 3.2: Фибоначијева стабла T_0, T_1, T_2 и T_3.

Може се показати и јаче тврђење.

Теорема 6. *За АВЛ стабло са n чворова висина h задовољава услов $h < 1.441\log_2 n$.*

Доказ. Кренимо од добијене рекурентне једначине за минимални број чворова АВЛ стабла висине h: $n_h = n_{h-1} + n_{h-2} + 1$. Важи $n_h + 1 = (n_{h-1} + 1) + (n_{h-2} + 1)$, за $h \geq 2$. Видимо да бројеви $n_h + 1$ задовољавају исту диференцну једначину као и Фибоначијеви бројеви F_h ($F_{h+2} = F_{h+1} + F_h$ за $h \geq 1$; $F_1 = F_2 = 1$). Узимајући у обзир једнакост по прва два члана низова $n_0+1 = F_3 = 2$ и $n_1+1 = F_4 = 3$, индукцијом се непосредно показује да важи $n_h = F_{h+3} - 1$. Полазећи од израза за општи члан Фибоначијевог низа

$$F_h = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^h - \left(\frac{1-\sqrt{5}}{2}\right)^h\right) \geq \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^h - 1\right)$$

добијамо

$$n_h \geq \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - 1\right) - 1 \geq \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - 1.45,$$

Ова неједнакост еквивалентна је са

$$h \leq \log_{(1+\sqrt{5})/2}(\sqrt{5}(n_h + 1.45)) - 3 < 1.441 \log_2 n_h \leq 1.441 \log_2 n$$

чиме је теорема доказана.

3.1.1 Операције са АВЛ стаблом

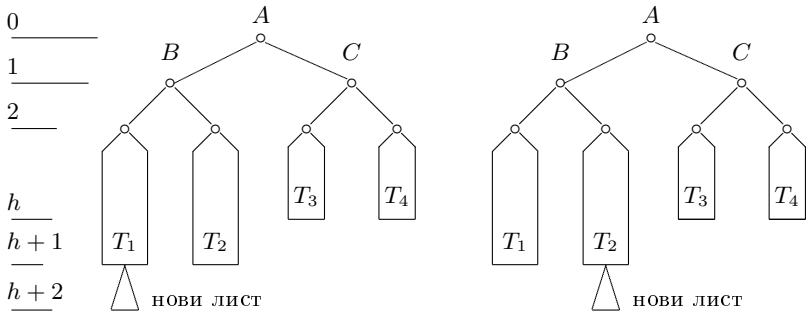
Размотримо сада како се после уметања, односно брисања елемента из АВЛ стабла, може интервенисати тако да резултат и даље буде АВЛ стабло.

Уметање елемента у АВЛ стабло

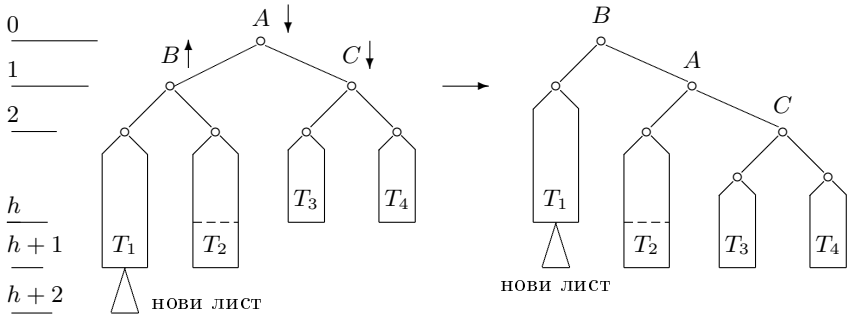
Приликом уметања новог елемента у АВЛ стабло поступа се најпре на начин уобичајен за бинарно стабло претраге: проналази се место чвору, па се у стабло додаје нови чвор са кључем једнаким задатом броју. Чворовима на путу који се том приликом прелазе одговарају разлике висина левог и десног подстабла (**фактори равнотеже**) из скупа $\{0, \pm 1\}$. Посебно је интересантан последњи чвор на том путу који има фактор равнотеже различит од нуле, тзв. **критични чвор**. Испоставља се да је приликом уметања броја у АВЛ стабло *довољно уравнотежити подстабло са кореном у критичном чвору* - у то ћемо се уверити када видимо детаље поступка уравнотежавања.

Претпоставимо да је фактор равнотеже у критичном чвору једнак 1 (видети слику 3.3). Нови чвор може да заврши у:

- десном подстаблу – у том случају подстабло коме је корен критични чвор остаје АВЛ
- левом подстаблу – у том случају стабло престаје да буде АВЛ и потребно је интервенисати. Према томе да ли је нови чвор додат левом или десном подстаблу левог подстабла, разликујемо два случаја, видети слику 3.3.
  - У првом случају се на стабло примењује **ротација**: корен левог подстабла B (видети слику 3.4) подиже се и постаје корен подстабла (коме је корен био критичан чвор), а остатак стабла преуређује се тако да стабло и даље остане БСП. Стабло T_1 се "подиже" за један ниво остајући и даље лево подстабло чвора B; стабло T_2 остаје на истом нивоу, али уместо десног подстабла B постаје лево подстабло A; десно подстабло A спушта се за један ниво. Пошто је A критичан чвор, фактор равнотеже чвора B је 0, па стабла T_1 и T_2 имају исту висину. Стабла T_3 и T_4 не морају имати исту висину, јер чвор C није на путу од критичног чвора до места уметања (слика 3.4). Нови корен подстабла постаје чвор B, са фактором равнотеже 0.

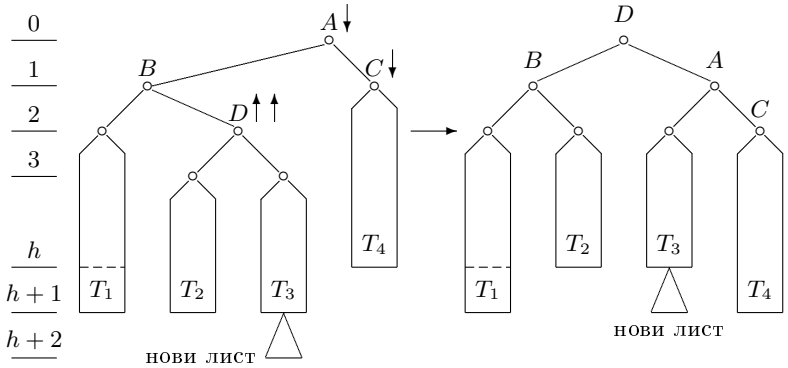


Слика 3.3: Уметања која ремете АВЛ својство стабла.



Слика 3.4: Уравнотежавање АВЛ–стабла после уметања — ротација.

– Други случај је компликованији; тада се стабло може уравнотежити **двоструком ротацијом**, видети слику 3.5. Нови чвор подстабла уместо критичног чвора постаје десни син левог сина критичног чвора.



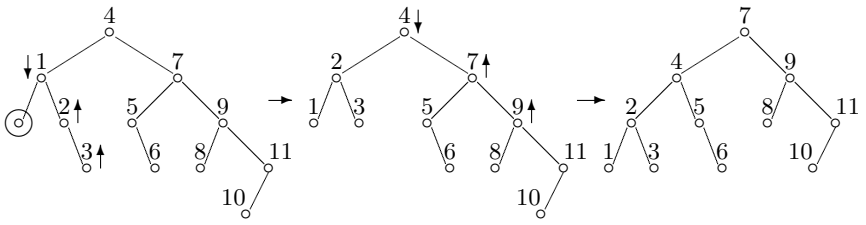
Слика 3.5: Уравнотежавање АВЛ–стабла после уметања — двострука ротација.

Запажамо да у оба случаја висина подстабла коме је корен критични чвор после уравнотежавања остаје непромењена. Уравнотежавање под-

стабла коме је корен критичан чвор због тога не утиче на остатак стабла. Уз сваки чвор стабла чува се његов фактор равнотеже, једнак разлици висина његовог левог и десног подстабла; за АВЛ стабло су те разлике елементи скупа $\{-1, 0, 1\}$. Уравнотежавање постаје неопходно ако је фактор неког чвора ± 1, а нови чвор се умеће на "погрешну" страну. После уравнотежавања висине изнад критичног чвора остају непромењене. Комплетан поступак уметања са уравнотежавањем дакле изгледа овако: идући наниже приликом уметања памти се последњи чвор са фактором равнотеже различитим од нуле – критичан чвор; кад се дође до места уметања, установљава се да ли је уметање на "доброј" или "погрешној" страни у односу на критичан чвор, а онда се још једном пролази пут уназад до критичног чвора, поправљају се фактори равнотеже и извршавају евентуалне ротације.

Брисање елемента из АВЛ стабла

Брисање је компликованије, као и код обичног БСП. У општем случају се уравнотежавање не може извести помоћу само једне или две ротације. На пример, да би се Фибоначијево стабло F_h са n чворова уравнотежило после брисања "лоше" изабраног чвора, потребно је извршити $h - 2$, односно $O(\log n)$ ротација (видети слику 3.6 за $h = 4$). У општем случају је граница за потребан број ротација $O(\log n)$. На срећу, свака ротација захтева константни број корака, па је време извршавања брисања такође ограничено одозго са $O(\log n)$. На овом месту прескочићемо детаље.



Слика 3.6: Уравнотежавање Фибоначијевог стабла T_4 после брисања чвора, помоћу две ротације.

3.1.2 Сложеност основних операција са АВЛ стаблом

АВЛ стабло је ефикасна структура података. Чак и у најгорем случају АВЛ стабла захтевају највише за 45% више упоређивања од оптималних стабала. Емпиријска испитивања су показала да се просечно тражење састоји од око $\log_2 n + 0.25$ поређења. Основни недостатак АВЛ стабала је потреба за додатним меморијским простором за смештање фактора равнотеже, као и чињеница да су програми за рад са АВЛ стаблима доста компликовани. Постоје и друге варијанте уравнотежених стабала претраге, на пример $2 - 3$ стабла, B-стабла и црвено-црна стабла.

3.2 Скип листе

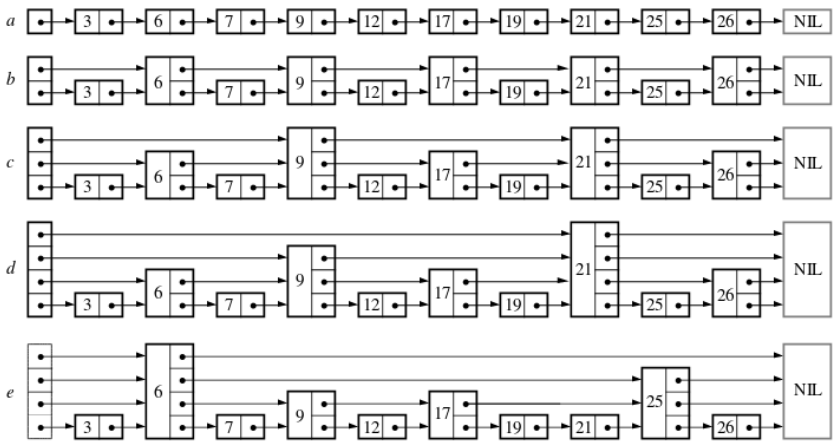
Уређена бинарна стабла се често користе за реализацију речника. Она се добро понашају уколико се елементи умећу случајним редоследом. Међутим, неки низови операција, као што је уметање елемената уређених растуће према вредности дегенеришу бинарно стабло и кваре његове перформансе. Алгоритми за рад са уравнотеженим (балансираним) стаблима мењају структуру стабла током извршавања основних операција да би се одржао услов уравнотежености и осигурале добре перформансе.

Скип листа (брза листа или листа са пречицама) је пробабилистичка алтернатива уравнотеженим стаблима, код које се уравнотежење постиже коришћењем генератора случајних бројева. Иако у најгорем случају скип листе имају лоше перформансе, ни један улаз не одговара конзистентно најгорем случају (слично алгоритму квиксорт када се пивот бира на случајан начин). Мало је вероватно да скип листа буде значајно неуравнотежена. Дакле, скип листа има својство уравнотежености налик уређеним бинарним стаблима изграђеним случајним уметањима елемената, али не захтева да уметања буду случајна.

Приликом рада са обичном листом понекад је потребно прегледати сваки чвор листе (слика 3.7а)). Уколико листа чува елементе у сортираном редоследу и сваки други чвор има и показивач на чвор два места испред њега у листи онда је максимални број чворова које треба испитати $\lceil n/2 \rceil + 1$ (где је са n означена укупна дужина листе); за тражење се користе прорећени показивачи док се не наиђе на чвор са кључем већим од траженог (слика 3.7б)). На пример, ако листа има 5 чворова и ми тражимо елемент који се налази на четвртом месту у листи, најпре бисмо испитали први чвор, закључили да је вредност елемента који тражимо већа од вредности у овом чвору и показивачем који прескаче по два чвора листе испитали трећи чвор листе, пошто је и његова вредност мања од тражене, наредни бисмо испитали пети чвор листе. Међутим, вредност петог чвора листе је већа од тражене, те бисмо из трећег чвора листе идући показивачем који повезује суседне чворове листе испитали четврти чвор листе и закључили да се тражена вредност налази на том месту. Дакле укупно бисмо анализирали 4 чвора листе што је једнако $\lceil 5/2 \rceil + 1$.

Ако би сваки четврти чвор поред тога имао и показивач ка чвору четири места испред њега, максимални укупан број чворова које треба анализирати био би $\lceil n/4 \rceil + 2$ (слика 3.7ц)). Ако сваки 2^i-ти чвор има показиваче на чворове $2^0, 2^1, \ldots, 2^i$ места испред њега у листи, максимални број чворова које треба анализирати се смањује на $O(\log_2 n)$, по цену дуплирања укупног броја показивача (слика 3.7д)). Оваква структура података би била ефикасна за претрагу, али би брисања и уметања била потпуно непрактична.

Назовимо чвор који има k показивача унапред **чвором нивоа** k. Уколико сваки 2^i-ти чвор има показивач на чвор 2^i места испред њега, онда су нивои чворова расподељени на следећи начин: 50% чворова је нивоа 1, 25% чворова је нивоа 2, 12.5% њих је нивоа 3, итд. Шта би се десило ако би нивои чворова били бирани на случајан начин, али у истој пропорцији (слика 3.7е))? Чворов i-ти показивач би уместо да показује на чвор 2^{i-1} места испред њега, показивао на наредни чвор нивоа i или већег. Уметања и брисања би захтевала само локалне измене. Ниво чвора изабран на случајан начин приликом креирања не би морао никада да се мења. Овакву структуру



Слика 3.7: Повезана листа са додатним показивачима

података називамо **скип листом**.

3.2.1 Операције са скип листом

Сада ћемо размотрити како се над скип листом изводе основне операције речника: претрага, уметање и брисање.

Сваки елемент је представљен чвором чији се степен бира на случајан начин приликом уметања. Чвор нивоа i има i показивача унапред, са индексима од 1 до i. Нивои су ограничени константом $MaxNivo$. Ниво листе је максимални ниво чвора листе (или 1 ако је листа празна). Глава листе (почетни елемент, који постоји и у празној листи) има показиваче на нивоима од 1 до $MaxNivo$. Показивачи унапред главе листе на нивоима већим од тренутно максималног нивоа листе показују на елемент NIL.

Иницијализација скип листе се састоји од два корака:

- алоцира се елемент NIL и даје му се вредност већа од свих дозвољених вредности за кључ,
- иницијализује се нова листа тако да је $nivo$ листе једнак 1 и сви показивачи главе листе показују на NIL.

Тражење елемента у скип листи

Тражење елемента у скип листи са кључем једнаким задатом броју a почиње на највишем нивоу листе. На том нивоу прате се показивачи док се не наиђе на кључ који је већи или једнак од a. У претходном чвору на том нивоу прелази се на следећи "нижи" показивач. Поступак се понавља на том и свим нижим нивоима. Кад се на основном нивоу наиђе на чвор са кључем једнаким a, претрага је завршена.

Ако се претпостави да сваки члан листе има поља k (кључ), $vrednost$ и низ $naredni$ поступак тражења описује следећи код.

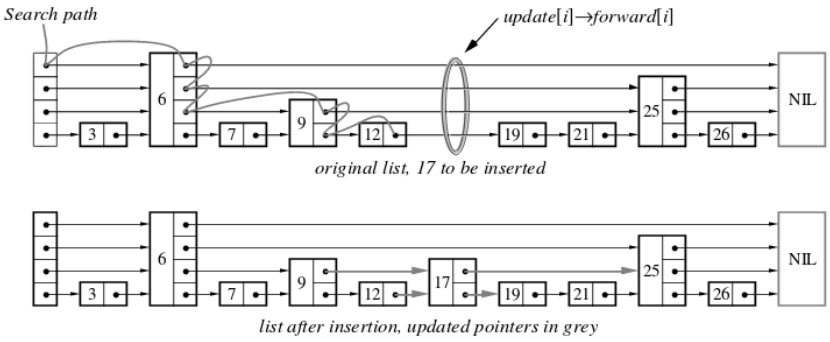
На пример, ако се у листи на слици 3.8 тражи број 12, полази се од чвора 6 и нивоа 4, па се наставља са истим чвором на нивоу 3 и нивоу 2,

Алгоритам Search$(L, kljuc)$;
Улаз: L (скип листа) и $kljuc$ (кључ елемента који се тражи).
Излаз: вредност елемента са датим кључем или *neuspeh* ако се елемент не налази у листи.
begin
    $x := L \rightarrow glava$
    for $i := L \rightarrow nivo$ **downto** 1 **do**
        while $x \rightarrow naredni[i] \rightarrow k < kljuc$ **do**
            $x := x \rightarrow naredni[i]$
    {инваријанта петље: x је последњи елемент нивоа i са кључем мањим од $kljuc$}
    $x := x \rightarrow naredni[1]$
    if $x \rightarrow k = kljuc$ **then return** $x \rightarrow vrednost$
    else return *neuspeh*
end

долази се до чвора 9 на нивоу 2, чвора 9 на нивоу 1, чвору 12 на нивоу 1. Ту се искаче из петље, при чему је у том тренутку x уперен на чвор 9 на нивоу 1. Наредно премештање води нас у чвор са траженим кључем 12.

Напомена: у приказаним примерима као вредности чворова коришћени су њихови кључеви. У општем случају кључеви се користе за уређење листе, а вредности у чворовима могу бити произвољне.

Брисање елемента из скип листе



Слика 3.8: Илустрација два корака од којих се састоји операција уметања вредности 17 у дату скип листу.

Да бисмо обрисали чвор из скип листе, потребно је да претражимо скип листу и да превежемо показиваче. Приликом тражења подаци неопходни за брисање чувају се у помоћном вектору *pom* дужине једнаке броју нивоа листе. Прецизније, $pom[i]$ садржи показивач на најдеснији чвор нивоа i или вишег који се налази лево од локације где треба извести брисање. На пример, ако из листе са слике 3.8 желимо да обришемо чвор са кључем 25, прво проналазимо тај чвор. Притом вредности $pom[4]$ и $pom[3]$ садрже показивач на чвор са вредношћу 6, вредност $pom[2]$ показивач на чвор са вредношћу 17, а $pom[1]$ показивач на чвор са вредношћу 21. Чвор 25 брише

се тако што се превезују показивачи на нивоима мањим или једнаким од његовог нивоа (3): показивачи у чворовима 6, 17 и 21 (на које показују елементи низа *pom*) добијају нове вредности, тако да показују на чворове редом NIL, NIL, 26.

Након сваког брисања, врши се провера да ли је обрисани чвор једини чвор са максималним нивоом у листи; ако јесте, смањује се максимални ниво листе.

Алгоритам Delete$(L, kljuc)$;
Улаз: L (скип листа) и $kljuc$ (кључ елемента који се брише).
begin
    $x := L \to glava$
    for $i := L \to nivo$ **downto** 1 **do**
        while $x \to naredni[i] \to k < kljuc$ **do**
            $x := x \to naredni[i]$
        $pom[i] := x$
    $x := x \to naredni[1]$
    if $x \to k = kljuc$ **then**
        for $i := 1$ **to** $L \to nivo$ **do**
            if $pom[i] \to naredni[i] \neq x$ **then break** {премашили смо степен чвора x}
            $pom[i] \to naredni[i] := x \to naredni[i]$
        $free(x)$
        while $L \to nivo > 1$ **and** $L \to glava \to naredni[L \to nivo] =$ **NULL do**
            $L \to nivo := L \to nivo - 1$
end

Уметање елемента у скип листу

Да бисмо уметнули чвор у скип листу, као и код брисања, потребно је да претражимо листу и да превежемо показиваче (слика 3.8). За превезивање показивача нам је опет потребан вектор *pom* који, као и у случају брисања елемента из листе, садржи показивач на најдеснији чвор нивоа i или вишег који се налази лево од локације где треба извести уметања.

Уколико уметање генерише чвор већег нивоа него што је претходни ниво листе, ажурирамо вредност нивоа листе и иницијализујемо одговарајуће делове низа *pom*.

Бирање нивоа чвора на случајан начин

Како на случајан начин изабрати ниво новог чвора? Претпоставимо да проценат p чворова нивоа i има показиваче нивоа $i + 1$ (најчешће се бира $p = 1/2$). Да би се одредио ниво новог чвора користи се генератор случајних бројева, који на излазу даје случајни број из интервала $[0, 1)$ са равномерном расподелом вероватноћа. Сваки пут када се на излазу из генератора добије број мањи од p (што се дешава са вероватноћом једнаком управо p), ниво се повећава за 1, сем ако већ није достигнут максимални ниво.

Овакав начин генерисања нивоа некада може бити неефикасан јер подразумева потенцијално већи број позива функције $random()$ (2 позива у просеку).

Алгоритам Insert$(L, kljuc, nova\_vrednost)$;
Улаз: L (скип листа), $kljuc$ (кључ елемента који се додаје) и $nova\_vrednost$ (вредност елемента који се додаје).
begin
    $x := L \rightarrow glava$
    for $i := L \rightarrow nivo$ **downto** 1 **do**
        while $x \rightarrow naredni[i] \rightarrow k < kljuc$ **do**
            $x := x \rightarrow naredni[i]$
        $pom[i] := x$
    $x := x \rightarrow naredni[1]$
    if $x \rightarrow k = kljuc$ **then** {већ постоји та вредност кључа, само ажурирамо вредност}
        $x \rightarrow vrednost := nova\_vrednost$
    else
        $nivo := SluchajniNivo()$ { на случајан начин бира се ниво новог чвора }
        if $nivo > L \rightarrow nivo$ **then** { ако је ниво новог чвора већи од текућег нивоа листе}
            for $i := L \rightarrow nivo + 1$ **to** $nivo$ **do**
                $pom[i] := L \rightarrow glava$
            $L \rightarrow nivo := nivo$
        $x := makeNode(nivo, kljuc, nova\_vrednost)$
        for $i := 1$ **to** $nivo$ **do**
            $x \rightarrow naredni[i] := pom[i] \rightarrow naredni[i]$
            $pom[i] \rightarrow naredni[i] := x$
end

Алгоритам SluchajniNivo();
Излаз: ниво чвора.
begin
    $nivo := 1$
    while $random() < p$ **and** $nivo < MaxNivo$ **do**
        $nivo := nivo + 1$
    return $nivo$
end

Уместо овога можемо искористити функцију $random\_int()$ која враћа случајан цео број у опсегу $[0, MAX\_INT]$ и вратити позицију првог постављеног бита у запису тог броја (ова имплементација подразумева да $MaxNivo$ није већи од броја битова у запису броја који враћа функција $random\_int()$).

Може се десити да у скип листи од 16 елемената генерисаних помоћу вредности $p = 1/2$ имамо 9 елемената нивоа 1, 3 елемента нивоа 2, 3 елемента нивоа 3 и један елемент нивоа 14. Ако бисмо претрагу вршили коришћењем претходно наведеног алгоритма, имали бисмо пуно празног хода. Где би требало започети претрагу? Спроведена анализа сугерише да би идеално било започети претрагу на нивоу $L(n)$ на коме очекујемо $1/p = 2$ чвора. Ово се дешава за ниво $L(n) = \log_{1/p} n$. Дакле, за вредност $MaxNivo$ ћемо у алгоритму бирати вредност $L(n)$.

Приметимо да ни за једну од наведених операција није потребно да у самом чвору чувамо информацију о његовом нивоу.

3.2.2 Сложеност основних операција са скип листом

Временима потребним за извршавање операција претраге, уметања и брисања доминира време потребно за налажење одговарајућег елемента у скип листи; код операција уметања и брисања постоји додатна цена пропорционална нивоу чвора који се умеће или брише. Време потребно за проналажење елемента је пропорционално дужини пута потраге. Анализираћемо пут потраге уназад, од чвора који смо тражили, тако да напредујемо ка вишим нивоима и налево. Означимо са $C(k)$ очекивану цену (тј. дужину) пута ако смо на нивоу k одозго. Тада је $C(0) = 0$, јер се при тражењу полази од највишег нивоа, нивоа 0. Подсетимо се да уколико можемо да се "попнемо" неки ниво, увек то и радимо јер тиме идемо "краћим" путем; само уколико не можемо да се крећемо навише идемо улево. Приликом процене очекиване дужине пута $C(k)$ претпостављамо да је скип листа бесконачне дужине.

- Са вероватноћом p у чвор где се сада налазимо доспели смо из истог чвора са вишег нивоа, са очекиваном дужином пута $C(k-1)$.
- Са вероватноћом $1-p$ у чвор где се сада налазимо доспели смо из чвора са истог нивоа, са очекиваном дужином пута $C(k)$.

Према томе, вредности $C(k)$ задовољавају диференцну једначину:

$$C(k) = (1-p)(1+C(k)) + p(1+C(k-1))$$

Сређивањем овог израза добијамо:
$C(k) = 1/p + C(k-1)$
односно у случају када је $p = 1/2$ очекивани број корака на сваком нивоу је 2. Даљим расписивањем израза за $C(k)$ добијамо $C(k) = k/p$

С обзиром на то да скип листа није бесконачна у неком моменту ћемо наићи на главу листе из које више нису могуће кретње улево већ само навише. Ако са $L(n) = \log_{1/p} n$ означимо просечни ниво листе дужине n, максимална вредност за k је $L(n) - 1$ јер се иницијално већ налазимо на нивоу 1 (на коме смо нашли чвор). Дакле, просечно време тражења је $C(k) = O(\log n)$. Повратак на највиши ниво нас не морамо нужно директно довести до главе скип листе, али очекивани број преосталих корака је мали и не утиче на укупну сложеност.

Скип листе не захтевају да се уз чвор листе чувају информације о уравнотежености (као код балансираних стабала), али зато захтевају чување додатних показивача (у просеку 2 показивача по чвору). Међутим, за многе примене скип листе су природнија репрезентација од стабала и воде једноставнијим алгоритмима. Доста су једноставније и за имплементацију од балансираних бинарних стабала. Такође, приликом операција уметања и брисања све измене су локалне, са изузетком промене максималног нивоа листе, те делују погодније за паралелизацију од балансираних бинарних претраживачких стабала.

3.3 Суфиксна стабла

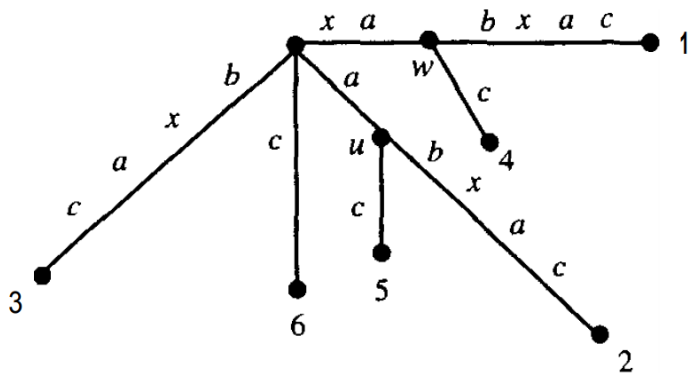
Суфиксно стабло је структура података којом се представља интерна структура ниске.

Стандардна примена суфиксних стабала је проблем тражења речи у тексту. Ако је дат текст T дужине n, након предобраде које је временске сложености $O(n)$, могуће је за дату непознату ниску S дужине m у времену $O(m)$ наћи неко њено појављивање у T или утврдити да се ниска S не налази у T. Дакле, предобрада је временске сложености пропорционалне дужини текста, али је након тога сложеност тражења S пропорционална дужини S, независно од дужине текста T. Ова граница се не може постићи алгоритмом КМП, чија је сложеност $O(n)$. С обзиром на то да n може бити велико у односу на m, овај резултат представља важно побољшање.

Први алгоритам линеарне сложености за конструкцију суфиксног стабла предложио је Вајнер (Weiner) 1973. године. Другачији, просторно ефикаснији алгоритам за формирање суфиксних стабала предложио је 1975. године Мекрајт (McCreight). Ми ћемо се у оквиру курса упознати са Укоиеновим (Ukonnen) алгоритмом који је такође линеарне временске сложености, али је једноставније разумети га.

Дефиниција 3.1 (Суфиксно стабло)**.** Суфиксно стабло *ниске S дужине m је коренско стабло за које важи:*

- *има тачно m листова који су нумерисани бројевима $1, 2, \ldots, m$;*
- *сваки унутрашњи чвор има барем два сина;*
- *свака грана је означена непразном подниском ниске S;*
- *никоје две гране из истог чвора не почињу истим карактером;*
- *када се надовежу све ознаке на путу од корена до листа са ознаком i, резултат је суфикс ниске S који почиње од позиције i, односно $S[i..m]$.*

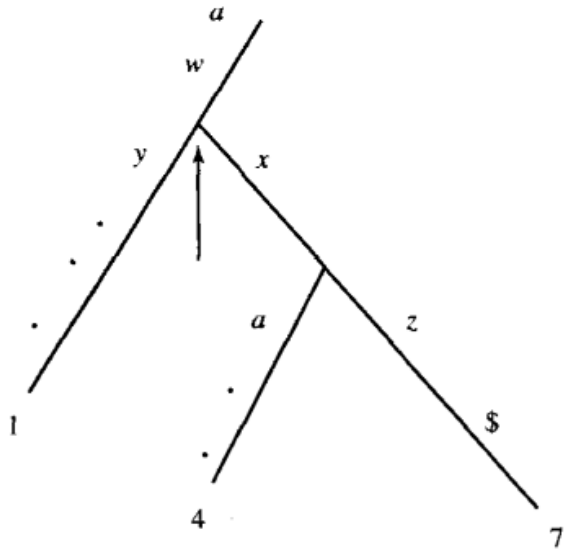


Слика 3.9: Суфиксно стабло ниске *xabxac*. Чворови са ознакама u и w су унутрашњи чворови.

На слици 3.9 приказано је суфиксно стабло ниске *xabxac*. Дефиниција суфиксног стабла ниске S не гарантује постојање суфиксног стабла за произвољну ниску S. Проблем се јавља онда када се неки суфикс ниске S поклопи са префиксом другог суфикса ниске S, јер тада се пут за први

суфикс не завршава у листу. На пример, у случају ниске $cxabxa$, постојао би пут од корена до суфикса $xabxa$ који би се завршавао у листу, али се зато пут до суфикса xa (као префикса суфикса $xabxa$) не би завршавао у листу. Да би се избегао овај проблем, претпостављамо да се последњи карактер ниске S не налази нигде другде унутар S. Ово се може постићи додавањем специјалног карактера који не припада азбуци језика у којем се формира ниска S. Тај карактер ћемо означавати са \$ и звати га *завршни* (или *терминирајући*) *карактер*.

Још један проблем који се једноставно може решити коришћењем суфиксног стабла јесте да се за дату реч P дужине m и текст T дужине n нађу *сва* појављивања P у T у времену $O(m+n)$. Решење преко суфиксних стабала би подразумевало формирање суфиксног стабла за ниску T у времену $O(n)$. Након тога бисмо пратили јединствени пут у суфиксном стаблу чији је префикс ниска P, све док не нађемо све карактере речи P или до тренутка када се испостави да се наредни карактер ниске P не може пронаћи ни на једном путу до неког листа. У другом случају можемо закључити да се ниска P нигде не јавља у тексту T, док у првом случају сваки лист у подстаблу са кореном у чвору у коме се десило последње поклапање карактера представља индекс од кога почиње појава речи P у тексту T. На пример, на слици 3.10 приказан је један фрагмент суфиксног стабла за ниску $awyawxawxz$. Реч $P = aw$ јавља се три пута у T на почетним позицијама $1, 4$ и 7.



Слика 3.10: Три појављивања ниске aw у ниски $awyawxawxz$. Њихове почетне позиције се налазе у листовима подстабла чвора где се завршава ниска aw.

3.3.1 Једноставни алгоритам за формирање суфиксног стабла

Да бисмо илустровали проблем формирања суфиксног стабла ниске S дужине m, описаћемо најпре решење проблема најједноставнијим алгоритмом. Овај једноставни алгоритам полази од стабла које садржи корен и једну грану, са ознаком суфикса $S[1..m]$ једнаког целој нисци S. Затим се за $i = 2, 3, \ldots, m$ у стабло редом укључује суфикс $S[i..m]$. Индуктивна хипотеза гласи: "Умемо да конструишемо стабло N_i које кодира све суфиксе ниске S који почињу на позицијама од 1 до i".

Базни случај је једноставан – стабло N_1 се састоји од једне гране означене ознаком $S\$$ која повезује корен и лист са ознаком 1. Стабло N_{i+1} се конструише од стабла N_i на следећи начин: полазећи од корена стабла N_i проналази се најдужи пут чија се ознака поклапа са префиксом ниске $S[i+1..m]\$$; под ознаком пута подразумева се конкатенација ознака грана које чине тај пут. Када се пронађе такав пут, он се завршава или у неком унутрашњем чвору w или унутар неке гране (u, v). Ако је унутар гране, онда се грана (u, v) "разбија" на две гране уметањем чвора w тачно након последњег карактера на грани који се поклопио карактером у $S[i+1..m]$ и тачно испред првог карактера на грани који се није поклопио. Нова грана (u, w) означава се делом ознаке гране (u, v) која се поклопила са префиксом ниске $S[i+1..m]$, а нова грана (w, v) се означава остатком ознаке гране (u, v). Затим, било да се нови чвор w формирао или да је неки чвор већ постојао на месту где се поклапање завршило, алгоритам креира нову грану $(w, i+1)$ која повезује чвор w и нови лист са ознаком $i+1$ и означава нову грану непоклопљеним делом суфикса $S[i+1..m]\$$.

Сложеност. Проналажење пута који одговара једном суфиксу траје $O(m)$, а завршна обрада тог суфикса је сложености $O(1)$. Ово се ради за сваки суфикс, па је према томе укупна временска сложеност алгоритма $O(m^2)$.

3.3.2 Уконенов алгоритам за формирање суфиксног стабла

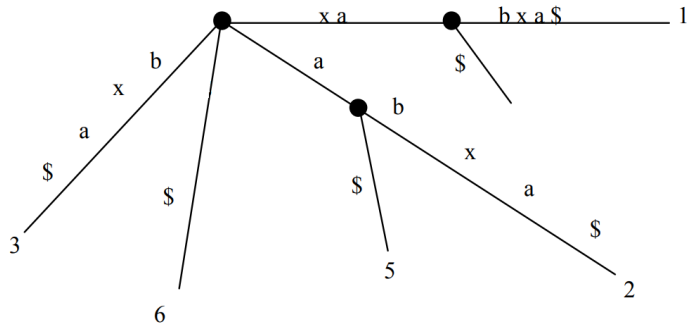
Еско Уконен је конструисао алгоритам за формирање суфиксног стабла у линеарном времену и он је добијен комбинацијом неколико лепих увида у структуру суфиксног стабла и неколико паметних имплементационих детаља. Током извршавања алгоритма формира се низ *имплицитних суфиксних стабала*, после чега се последње од добијених имплицитних суфиксних стабала претвара у право суфиксно стабло ниске S.

Дефиниција 3.2 (Имплицитно суфиксно стабло)**.** Имплицитно суфиксно стабло *ниске S је стабло добијено од суфиксног стабла ниске $S\$$ брисањем свих појава завршног карактера \$ са ознака грана стабла, након тога брисањем свих грана са празном ознаком и, коначно, брисањем свих чворова који немају бар два сина.*

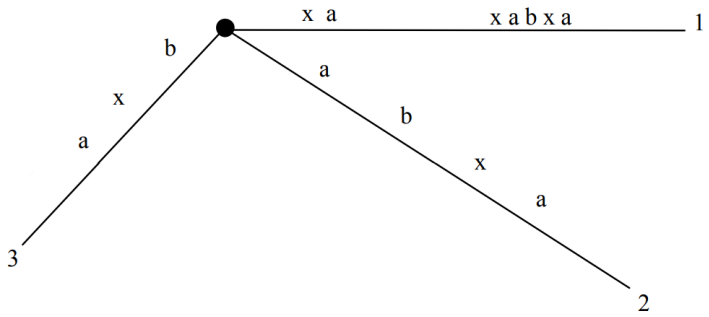
Имплицитно суфиксно стабло за префикс $S[1..i]$ ниске S означаваћемо са T_i. Стабло T_i се добија применом поступка из дефиниције 3.2 на суфикс $S[1..i]$.

Имплицитно суфиксно стабло за произвољну ниску S има мање од $m = |S|$ листова ако и само ако барем један од суфикса ниске S представља префикс неког другог суфикса. Завршни карактер \$ се додаје на крај ниске

S управо да би се избегла ова ситуација. Ако се сама ниска S завршава карактером који се не појављује нигде другде у S, онда имплицитно суфиксно стабло ниске S има лист за сваки суфикс, па је самим тим право суфиксно стабло.



Слика 3.11: Суфиксно стабло ниске $xabxa\$$.



Слика 3.12: Имплицитно суфиксно стабло ниске $xabxa$.

На пример, посматрајмо суфиксно стабло ниске $xabxa\$$ приказано на слици 3.11. Суфикс xa је префикс суфикса $xabxa$, и слично је ниска a префикс суфикса $abxa$. Зато су у суфиксном стаблу за $xabxa$ гране које воде до листова 4 и 5 означене само карактером \$. Одбацивањем тих грана производе се два нова чвора која имају само једног сина, па се онда и они одбацују. Резултат је имплицитно суфиксно стабло ниске $xabxa$ приказано на слици 3.12.

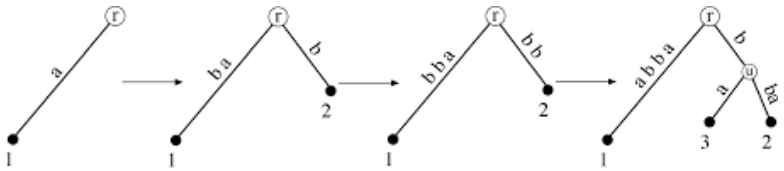
Иако имплицитно суфиксно стабло не мора да има лист за сваки суфикс, оно ипак кодира све суфиксе ниске S – сваки суфикс се добија надовезивањем карактера на неком путу од корена имплицитног суфиксног стабла. Ипак, ако се пут не заврши у листу, не постоји ознака која означава крај пута. Због тога су имплицитна суфиксна стабла сама по себи мање информативна од суфиксних стабала.

Уконенов алгоритам конструише имплицитно суфиксно стабло T_i за сваки префикс $S[1..i]$ ниске S, полазећи од T_1, све док се не формира T_m. Суфиксно стабло ниске S се конструише од последњег имплицитног суфиксног стабла T_m и време извршавања целокупног алгоритма је $O(m)$.

Најпре ћемо приказати метод који захтева $O(m^3)$ корака да се формирају сва стабла T_i, а након тога приказаћемо оптимизације у имплементацији Укоченовог алгоритма које омогућавају да алгоритам постане линеарне временске сложености.

Опис Укоченовог алгоритма

Размотримо случај ниске $S = abba$. Циљ нам је да одредимо имплицитна суфиксна стабла T_i за све префиксе $S[1..i]$, $i = 1, 2, 3, 4$ ниске S. На слици 3.13 приказана су добијена имплицитна стабла за све префиксе ниске S.



Слика 3.13: Имплицитна суфиксна стабла свих префикса ниске *abba*.

Укоченов алгоритам се састоји од m *фаза*. У $(i+1)$-ој фази стабло T_{i+1} се формира на основу стабла $T_i, i = 1, 2, \ldots, m$. Фаза $i+1$ састоји се од $i+1$ *продужења*, по једног за сваки од $i+1$ суфикса ниске $S[1..i+1]$. Користи се чињеница да се за $j = 1, 2, ..., i$ суфикс $S[j..i+1]$ добија продужавањем суфикса $S[j..i]$ (који је већ у стаблу) карактером $S[i+1]$.

У продужењу j фазе $i+1$, проналазимо место суфиксу $S[j..i+1], j = 1, 2, \ldots, i+1$. У имплицитном стаблу T_i формираном у претходној фази проналази се крај пута (од корена) са ознаком $S[j..i]$. Затим се подниска $S[j..i]$ као ознака пута продужује додавањем карактера $S[i+1]$ на њен крај, осим уколико се карактер $S[i+1]$ већ не налази тамо (у наставку ознаке истог пута). Продужење $i+1$ фазе $i+1$ продужује празан суфикс ниске $S[1..i]$, односно, "умеће" ниску која се састоји од једног карактера, $S[i+1]$, у стабло (осим у ситуацији када се тај карактер већ налази у стаблу). Овај алгоритам може се описати кодом са слике 3.14.

Правила за продужавање суфикса. У оквиру овог алгоритма неопходно је прецизирати како се изводи продужавање суфикса. Нека је $S[j..i] = \beta$ суфикс ниске $S[1..i]$. У продужењу j, када алгоритам пронађе крај суфикса β у тренутном стаблу, врши се продужавање ниске β да би се и нови суфикс $\beta S[i+1]$ нашао у стаблу. При томе су могућа три случаја, односно продужење се изводи применом једног од наредна три правила:

Правило 1: У тренутном стаблу, пут β се завршава у листу. Другим речима, последња грана пута означеног са β је грана e која се завршава у листу. Да би се стабло продужило, карактер $S[i+1]$ се додаје на крај ознаке гране e.

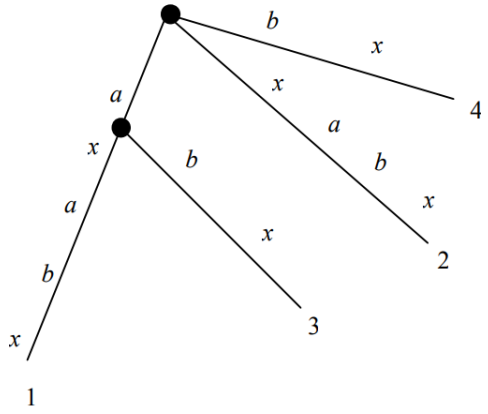
Правило 2: Ниједан пут од краја ниске β не почиње карактером $S[i+1]$, али барем један означени пут наставља од краја ниске β. У овом случају, формира се нова грана од краја ниске β и означава се карактером $S[i+1]$. Ако се β завршава унутар гране, на том месту се формира

Алгоритам Ukonenov_algoritam(S);
Улаз: S (ниска од које се конструише суфиксно стабло), m (дужина ниске).
Излаз: суфиксно стабло ниске S.
begin
Конструиши стабло T_1;
for $i := 1$ **to** $m - 1$ **do**
{започињемо фазу $i + 1$}
for $j := 1$ **to** $i + 1$ **do**
{започињемо продужење j}
Пронађи пут од корена са ознаком $S[j..i]$ у тренутном стаблу
if $S[i + 1]$ се не налази иза краја пута **then**
{Овиме се постиже да је ниска $S[j..i + 1]$ у стаблу}
Продужи пут додавањем карактера $S[i + 1]$
end

Слика 3.14: Кôд једноставног алгоритма за формирање суфиксног стабла сложености $O(m^3)$.

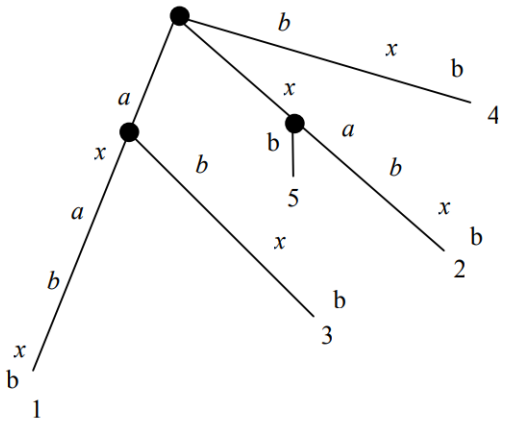
нови чвор. Листу на крају нове гране се придружује број j (пошто лист одговара суфиксу који почиње од индекса j).

Правило 3: Неки пут од краја ниске β почиње карактером $S[i+1]$. У овом случају ниска $\beta S[i + 1]$ се већ налази у стаблу, тако да није потребно урадити ништа (присетимо се да у имплицитном суфиксном стаблу крај суфикса не мора да буде експлицитно означен).



Слика 3.15: Имплицитно суфиксно стабло ниске $axabx$ пре додавања шестог карактера b.

На пример, посматрајмо имплицитно суфиксно стабло ниске $S = axabx$ приказано на слици 3.15. Прва четири суфикса се завршавају у листовима, али се једнокарактерски суфикс x завршава унутар гране. Када се шести



Слика 3.16: Продужено имплицитно суфиксно стабло ниске $axabx$ након додавања шестог карактера b.

карактер b додаје ниски, прва четири суфикса се продужују применом правила 1, пети суфикс се продужује применом правила 2, а шести суфикс применом правила 3. Резултат је приказан на слици 3.16.

Имплементација и убрзање. Коришћењем правила за продужавање суфикса, после проналажења суфикса β ниске $S[1..i]$ у тренутном стаблу, потребан је само константан број корака за примену правила за продужавање (односно, да би се обезбедило да се суфикс $\beta S[i+1]$ налази у стаблу). Због тога је основни проблем у реализацији Укоеновог алгоритма проналажење свих i суфикса ниске $S[1..i]$.

Најједноставнији приступ омогућује проналажење било ког суфикса β у $O(|\beta|)$ корака, проласком пута од корена тренутног стабла. Дакле, продужење j фазе $i+1$ би захтевало $O(i+1-j) = |S[i..j]|$ корака. Пошто је укупан број продужења једнак $i+1$, стабло T_{i+1} се формира од стабла T_i у времену $O(i^2)$, а сложеност формирања стабла T_m је $O(m^3)$. На први поглед није јасно зашто смо уопште разматрали овај алгоритам сложености $O(m^3)$, када већ знамо једноставан алгоритам сложености $O(m^2)$. Међутим, методички је алгоритам сложености $O(m^3)$ веома важан, јер представља полазну тачку за развој Укоеновог алгоритма линеарне временске сложености.

Убрзање са $O(m^3)$ на $O(m)$ може се извести на основу неколико запажања и имплементационих трикова. Сваки трик за себе делује као разумна хеуристика за убрзање алгоритма, али њихова појединачна примена не повлачи смањење сложености у најгорем случају. Међутим, уколико се користе заједно, омогућавају извршење алгоритма у линеарном времену у најгорем случају.

Дефиниција 3.3 (Суфиксна веза)**.** *Нека $x\alpha$ означава произвољну ниску, где је x карактер и α (евентуално празна) ниска. Ако за унутрашњи чвор v са ознаком пута $x\alpha$ постоји неки други чвор $s(v)$ са ознаком пута α, онда се показивач од v до $s(v)$ назива* суфиксна веза.

Суфиксну везу означаваћемо уређеним паром $(v, s(v))$. У специјалном случају када је α празна ниска, суфиксна веза из унутрашњег чвора са

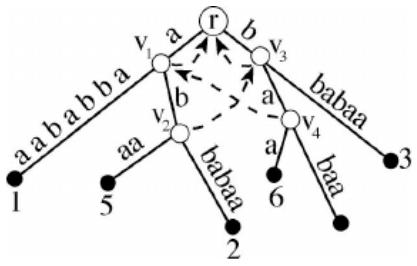
ознаком пута $x\alpha = x$ води ка корену стабла. Корен се не сматра унутрашњим чвором и не постоји суфиксна веза која полази из њега.

Из дефиниције суфиксне везе не следи да сваки унутрашњи чвор имплицитног суфиксног стабла има суфиксну везу која полази из њега. Испоставља се да је ово тврђење ипак тачно.

Лема 3.1 (Постојање суфиксних веза у стаблу)**.** *Ако се тренутном стаблу у продужењу j фазе $i+1$ дода нови унутрашњи чвор v са ознаком пута $x\alpha$, онда важи једно од наредна два тврђења:*

- *пут од корена са ознаком α се већ завршава у унутрашњем чвору тренутног стабла*
- *биће формиран унутрашњи чвор на крају ниске α (применом правила за продужење) у (наредном) продужењу $j+1$ фазе $i+1$.*

Доказ. Ниска која се разматра у фази $i{+}1$ је $S[1..i{+}1]$. Разматрамо правило 2 за продужавање суфикса, јер се једино приликом примене овог правила може формирати нови унутрашњи чвор: током продужења j, односно разматрања суфикса $S[j..i+1]$, унутрашњи чвор је или већ постојао или се креира додавањем карактера $S[i+1]$ на пут $S[j..i]$. Суфиксна веза треба да од тог чвора води ка чвору за ознаком пута $S[j{+}1..i{+}1]$, који или већ постоји, или ће бити креиран након продужења суфикса $S[j+1..i]$ карактером $S[i+1]$ током продужења $j+1$ фазе $i+1$.



Слика 3.17: Суфиксне везе за имплицитно суфиксно стабло ниске *aabbabaa*.

Теорема 3.1 (Постојање суфиксних веза у Укоиеновом алгоритму)**.** *У Укоиеновом алгоритму било који новоформирани унутрашњи чвор до краја наредног продужења добиће суфиксну везу која полази од њега.*

Доказ. Докажимо ово тврђење индукцијом по редном броју фазе i. Тврђење тривијално важи за T_1, јер не постоје унутрашњи чворови. Претпоставимо да тврђење важи за фазу i, односно за стабло T_i. За унутрашње чворове креиране до претпоследњег продужења тврђење важи према леми 3.1. Размотримо последње, $(i+1)$-во продужење. У њему се додаје само знак $S[i+1]$ (правилом 1 или 3) без додавања нових унутрашњих чворова.

Теорема 3.1 утврђује важну чињеницу о имплицитним стаблима и специјално о суфиксним стаблима. Можемо је формулисати и на други начин.

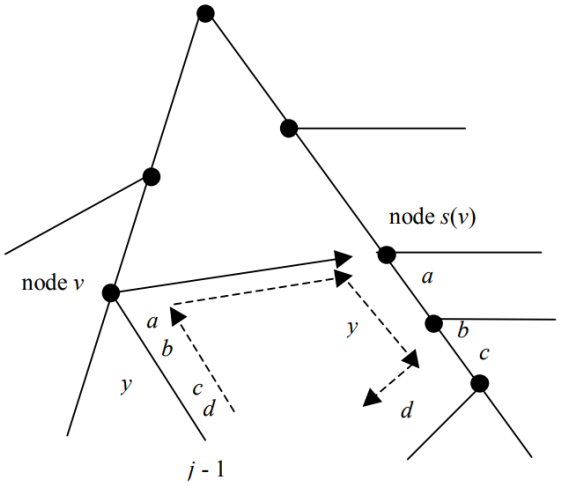
Теорема 3.2 (Постојање суфиксних веза у имплицитном суфиксном стаблу)**.** *У произвољном имплицитном суфиксном стаблу T_i, ако унутрашњи чвор има ознаку пута $x\alpha$, тада постоји чвор $s(v)$ стабла T_i са ознаком пута α.*

На основу теореме 3.1, сви унутрашњи чворови у текућем стаблу имаће суфиксне везе које полазе из њих, са изузетком последњег додатог унутрашњег чвора, који ће добити своју суфиксну везу до краја наредног продужења. Сада ћемо приказати како се суфиксне везе могу искористити да убрзају имплементацију.

Присетимо се да у фази $i+1$ алгоритам лоцира суфикс $S[j..i]$ ниске $S[1..i]$, за свако j од 1 до $i+1$ (за $j = i+1$ то је празан суфикс). У оквиру једноставног алгоритма ово се изводи праћењем ниске $S[j..i]$ полазећи од корена тренутног стабла. Суфиксне везе могу да скрате пут који се прелази и да на тај начин продужење учине ефикаснијим. Размотримо редом продужења:

- $j = 1$ (продужење 1): с обзиром на то да је $S[1..i]$ најдужа ниска кодирана у стаблу T_i, тражење ниске $S[1..i]$ се мора завршити у неком листу стабла T_i. То чини једноставним налажење краја тог суфикса (при формирању стабла можемо да чувамо показивач на лист до кога води пут са ознаком $S[1..i]$). Његово продужење врши се применом правила 1. Дакле, прво продужење било које фазе је специјалан случај и користи $O(1)$ корака, с обзиром на то да алгоритам садржи показивач на чвор са ознаком једнаком тренутној нисци.

- $j = 2$ (продужење $S[2..i]$ у $S[2..i+1]$): нека је ниска $S[1..i]$ (која се завршава у листу 1 стабла T_i) облика $x\alpha$, где је x карактер и α (евентуално празна) ниска, и нека је $(v, 1)$ грана стабла која улази у лист са ознаком 1 (обратимо пажњу да ова грана не мора бити означена само последњим карактером ове ниске већ може бити означена већим бројем карактера). Алгоритам треба да пронађе крај ниске $S[2..i] = \alpha$ у тренутном стаблу које се формира на основу стабла T_i. Кључ је у томе да је чвор v или корен или унутрашњи чвор стабла T_i. Ако је корен, онда да би се пронашао крај α прелази се пут од корена пратећи α као у једноставном алгоритму. Ако је пак v унутрашњи чвор, онда по теореми 3.2 (с обзиром да је v у стаблу T_i) чвор v има суфиксну везу до чвора $s(v)$. Даље, с обзиром да $s(v)$ има ознаку пута која је префикс ниске α, крај ниске α мора да се налази у подстаблу са кореном $s(v)$. Због тога приликом тражења краја ниске α у стаблу, алгоритам не мора да прелази читав пут од корена, већ може да крене од чвора $s(v)$. Ово је главни разлог за коришћење суфиксних веза у алгоритму. Прецизније, нека је γ ознака гране $(v, 1)$. Да би се пронашао крај α, треба се вратити из листа са ознаком 1 у чвор v, пратити суфиксну везу од v до $s(v)$, а затим проћи од чвора $s(v)$ низ стабло (тај пут може да се састоји од једне или више грана) путањом означеном ниском γ. Крај тог пута одређен је завршетком ниске α (видети слику 3.18). На крају пута са ознаком α стабло се ажурира коришћењем одговарајућег правила за продужење. Овим су у потпуности описана прва два продужења фазе $i+1$.



Слика 3.18: Продужење префикса $S[j..i]$ у префикс $S[j..i+1]$ у фази $i+1$: потребно је попети се највише једном граном (са ознаком γ; на слици је $\gamma = abcd$) од краја пута означеног $S[j-1..i]$ до чвора v; затим се прати суфиксна веза од v до $s(v)$; након тога пролази се од чвора $s(v)$ низ стабло путем означеним γ, чиме је у стаблу пронађена ниска $S[j..i]$. На крају се примењује одговарајуће правило продужења да би се уметнуо суфикс $S[j..i+1]$.

- Да би се продужила ниска $S[j..i]$ у ниску $S[j..i+1]$ за $j > 2$, користи се слична општа идеја. Да би се у стаблу пронашла ниска $S[j..i]$, полази се од краја ниске $S[j-1..i]$ (којој је она префикс), пење се највише једном граном до корена или унутрашњег чвора v који има суфиксну везу из себе; нека је γ ознака те гране. Под претпоставком да v није корен, пролази се суфиксном везом од v до $s(v)$; наставља се низ стабло од $s(v)$ пратећи пут са ознаком γ до краја $S[j..i]$. Коначно, врши се продужавање суфикса да се добије суфикс $S[j..i+1]$ коришћењем неког од правила за продужавање суфикса. Постоји једна мала разлика између продужења за $j > 2$ и прва два продужења. Уопштено говорећи, крај ниске $S[j-1..i]$ може бити у чвору који и сам има суфиксну везу која полази из њега (ако је то унутрашњи чвор), и у том случају алгоритам прати ту суфиксну везу. Приметимо да чак и када се правило 2 примењује у продужењу $j-1$ (тако да се крај $S[j-1..i]$ налази у новоформираном унутрашњем чвору w), ако отац чвора w није корен, онда отац чвора w већ има суфиксну везу која полази из њега на основу теореме 3.1. Због тога алгоритам у току продужења j никада на путу ка корену не прелази више од једне гране.

Изведена анализа омогућује да се прецизира алгоритам за продужење $S[j..i]$ у $S[j..i+1]$ у фази $i+1$:

- Полазећи од листа коме у стаблу одговара суфикс $S[j-1..i]$ на путу ка корену пронаћи први унутрашњи чвор v; чвор v или има суфиксну везу или је корен. Ово захтева пењање уз највише једну грану од

краја $S[j-1..i]$ у тренутном стаблу. Нека је γ (евентуално празна) ниска на грани којом се из v наставља пратећи $S[j-1..i]$.

- Ако v није корен, онда треба проћи суфиксном везом од v до $s(v)$ и затим проћи путем кроз стабло од $s(v)$ пратећи пут са ознаком γ. Ако је v корен, онда треба проћи путем ниске $S[j..i]$ од корена (као у једноставном алгоритму). Тиме је у стаблу пронађен крај ниске $S[j..i]$.
- Коришћењем одговарајућег правила за продужење суфикса, обезбедити да се ниска $S[j..i+1]$ налази у стаблу.
- Ако је формиран нови унутрашњи чвор w у продужењу $j-1$ (коришћењем правила 2), онда по теореми 3.1 ниска α мора да се заврши у чвору $s(w)$, чвору до којег долази суфиксна веза из чвора w. Формирати суфиксну везу $(w, s(w))$ од чвора w до чвора $s(w)$.

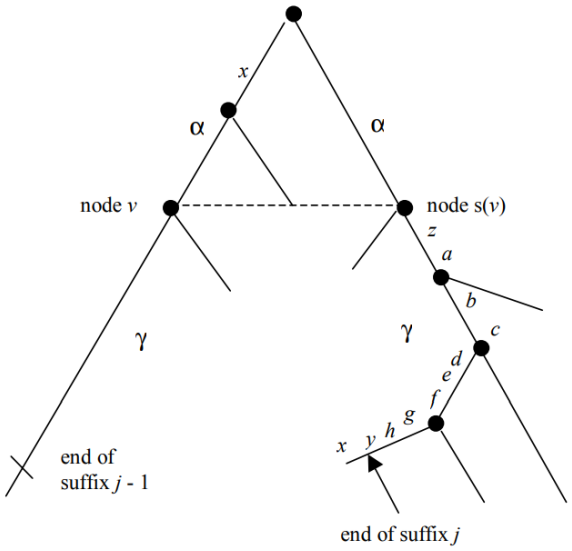
Под претпоставком да алгоритам чува показивач до тренутне целе ниске $S[1..i]$, прво продужење фазе $i+1$ не захтева никакво пењање уз, нити силазак низ стабло. Такође, прво продужење фазе $i+1$ увек примењује правило 1.

Коришћење суфиксних веза је очигледно практично побољшање у односу на полазак од корена у сваком продужењу, као што се ради у једноставном алгоритму. Међутим, ово побољшање није довољно да поправи брзину алгоритма у најгорем случају. У наставку ћемо описати имплементациони трик који смањује сложеност алгоритма у најгорем случају на $O(m^2)$.

У другом кораку продужења $j+1$ алгоритам силази низ стабло од чвора $s(v)$ пратећи пут са ознаком γ. Присетимо се да такав пут сигурно постоји од $s(v)$. Директно имплементиран, пролазак кроз пут са ознаком γ захтева $O(|\gamma|)$ корака, односно, пропорционалан је броју карактера ознаке γ. Ефикасније је приликом праћења ниске у стаблу прелазити гране за $O(1)$ (уместо за време пропорционално дужини ознаке гране). То се постиже преласком у наредни чвор преко гране и померањем позиције у нисци која се прати за број једнак дужини ознаке гране. Та идеја, тзв. *прескок са одбројавањем* (енг. skip/count) има за резултат да је сложеност преласка пута пропорционална броју грана на путу, уместо збиру дужина ознака грана на путу.

Трик 1 (Прескок са одбројавањем). Нека g означава број карактера ознаке γ. Присетимо се да никоје две ознаке грана које излазе из чвора $s(v)$ не могу да почну истим карактером, односно да први карактер h ознаке γ мора да се појави као први карактер на тачно једној грани која излази из $s(v)$. Нека g' означава број карактера на тој грани. Ако је $g > g'$, онда алгоритам нема потребе да чита ниједан други карактер на тој грани; треба да једноставно дође до чвора на крају те гране. Поред тога, ажурирају се вредности g и h, тако што се g замени са $g - g'$, а h се замени првим карактером ниске g'. Поступак се наставља на исти начин бирањем излазне гране чија ознака почиње карактером h. Уопште, када алгоритам пронађе наредну грану на путу, он упоређује тренутну вредност g и број карактера g' на тој грани. Када је $g > g'$, алгоритам прелази до чвора на крају те гране, замењује g са $g - g'$, замењује h карактером чији је индекс у нисци

већи за g', пронађи грану чији се први карактер поклапа са карактером h ознаке γ и понављај поступак. Када се достигне грана у којој је $g \leq g'$, онда алгоритам прелази до карактера са редним бројем g на тој грани и завршава се, пошто се пут са ознаком γ из чвора $s(v)$ завршава на тој грани тачно g карактера од почетка ознаке гране (видети слику 3.19).



Слика 3.19: Прескок са одбројавањем. У фази $i+1$, подниска $\gamma = zabcdefghy$ има дужину 10. Постоји копија подниске γ која почиње од чвора $s(v)$; она се завршава три карактера од почетка ознаке на последњој грани, након што се изведу четири прескакања грана.

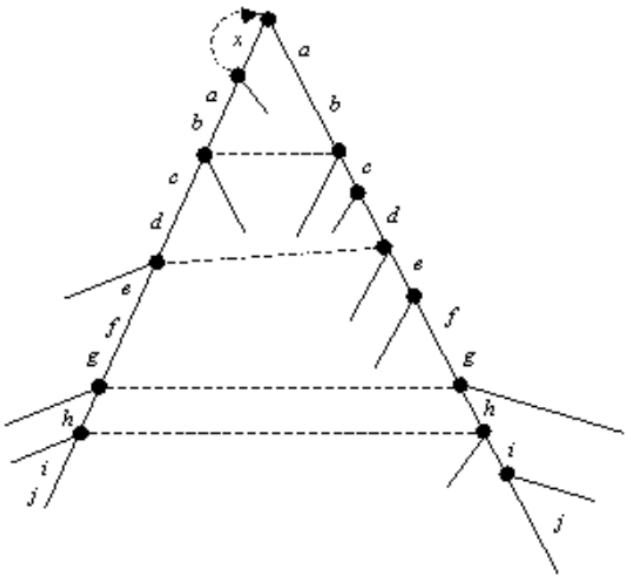
Под претпоставком да се зна дужина ознаке сваке гране стабла и да се карактер ниске S са задатим индексом пронађи за време $O(1)$, ефекат коришћења прескока са одбројавањем је прелажење гране при праћењу ознаке γ за време $O(1)$. Укупно време за проналажење пута који одговара нисци γ пропорционално је броју чворова на путу уместо броју карактера на њему.

Дефиниција 3.4 (Дубина чвора)**.** Дубина чвора *u је број чворова на путу од корена до u.*

Теорема 3.3 (Дубина чвора на крају суфиксне везе)**.** *Нека је* $(v, s(v))$ *произвољна суфиксна веза која се пролази у Уконеновом алгоритму. У тренутку када се користи суфиксна веза, дубина чвора* v *је за највише један већа од дубине чвора* $s(v)$*.*

Доказ. Размотримо путеве са ознакама $S[j..i{+}1]$ и $S[j{+}1..i{+}1]$ и унутрашњи чвор v на путу са ознаком $S[j..i+1]$. Кључно запажање је да из сваког претка чвора v води суфиксна веза ка јединственом претку чвора $s(v)$ који је такође унутрашњи чвор. Ово важи за све унутрашње чворове осим за први карактер пута $S[i..j+1]$ чија суфиксна веза води ка корену. Због ове повезаности унутрашњих чворова на два разматрана пута важи да дубина

чвора v може за највише један бити већа од дубине чвора $s(v)$. Илустрација овог доказа приказана је на слици 3.20.



Слика 3.20: За сваки чвор v на путу са ознаком $x\alpha$, одговарајући чвор $s(v)$ налази се на путу са ознаком α. При томе, ако је дубина чвора v једнака m, онда дубина чвора $s(v)$ може да буде $m-1$, m или већа од m. На пример, чвор са ознаком xab има дубину два, док је дубина чвора са ознаком ab једнака један. Слично, чвор са ознаком $xabcdefg$ има дубину пет, док је дубина чвора са ознаком $abcdefg$ једнака шест.

Дефиниција 3.5 (Тренутна дубина алгоритма)**.** *У току извршавања алгоритма* тренутна дубина алгоритма *је дубина чвора који је последњи посећен у алгоритму.*

Теорема 3.4 (Линеарност фазе Укоиеновог алгоритма)**.** *Коришћењем прескока са одбројавањем било која фаза Укоиеновог алгоритма захтева $O(m)$ корака.*

Доказ. У фази i се $i+1$ пут ради продужење суфикса. У току једног продужења, алгоритам прелази највише једну грану увис, прелази суфиксну везу, спушта се за одређени број чворова, примењује неко од правила за продужење суфикса и евентуално додаје нову суфиксну везу. Пошто смо већ установили да све остале операције сем спуштања трају $O(1)$, остаје да размотримо како се мења тренутна дубина чвора у току једне фазе.

Кретање увис у току било ког продужења смањује тренутну дубину чвора највише за један (пошто се на путу увис пролази највише један чвор). Прелазак преко суфиксне везе може да смањи тренутну дубину највише за још један (теорема 3.3), а свака грана пређена на путу наниже повећава тренутну дубину чвора. Према томе, у току целе фазе се дубина чвора може смањити за један највише $2m$ пута. Пошто ни један чвор нема

дубину већу од m, укупан број инкрементирања тренутне дубине чвора у току целе фазе не може да буде већи од $3m$. Према томе, укупан број грана пређених приликом кретања наниже у току целе фазе није већи од $3m$. Ако се користе прескоци са одбројавањем, прелазак гране траје $O(1)$, па је укупна сложеност дела алгоритма који одговара кретањима наниже $O(m)$.

Претходна теорема има за директну последицу наредно тврђење.

Теорема 3.5 (Квадратна сложеност Укоченовог алгоритма)**.** *Укоченов алгоритам се може имплементирати коришћењем суфиксних веза тако да захтева $O(m^2)$ корака.*

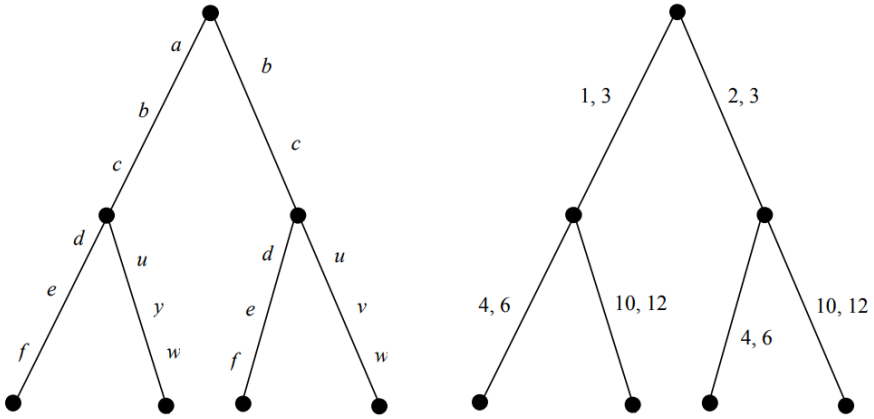
Сложеност $O(m^2)$ добија се множењем границе сложености за једну фазу алгоритма, $O(m)$, бројем фаза у алгоритму, m. Ова груба процена сложености последица је независне процене сложености произвољне фазе. Побољшање се постиже тако што се у анализи разматрају и преласци између фаза.

Важан детаљ у реализацији алгоритма. Циљ је постићи да временска сложеност алгоритма буде $O(m)$. На том путу постоји једна очигледна препрека: простор који заузима суфиксно стабло је $\Theta(m^2)$, јер ознаке грана могу да буду дужине $\Theta(m)$, а број грана је $\Theta(m)$. Због тога било који алгоритам за формирање суфиксног стабла мора да има сложеност бар $\Theta(m^2)$. Да би се дошло до алгоритма сложености $O(m)$, мора се обезбедити да ознаке грана заузимају простор $O(1)$.

Испоставља се да таква, једноставна алтернативна шема представљања ознака грана постоји. Уместо експлицитног записивања подниске на грани, може се користити уређен пар индекса на грани, који означавају почетну и крајњу позицију те подниске у S (видети слику 3.21). С обзиром да алгоритам чува копију ниске S, он је у стању да у константном времену пронађе било који карактер у S. Због тога је могуће описати било који алгоритам рада са суфиксним стаблом као да су ознаке грана представљене експлицитно, а затим имплементирати алгоритам са само константним бројем карактера на свакој грани. Описана шема назива се *компресија ознака грана*.

На пример, када се у оквиру Укоченовог алгоритма прати подниска која одговара преласку једне гране, користи се пар индекса записаних на грани да се из S прочитају карактери те ниске и затим се ти карактери упоређују са карактерима ниске. Правила за продужење се такође једноставно реализују коришћењем компресије ознака грана. Када се у фази $i{+}1$ примени правило 2, новоформирана грана се означава паром индекса $(i+1, i+1)$, а када се примени правило 1 (ка листу), промени се пар индекса на тој грани са (p, q) на $(p, q+1)$. Индукцијом се лако показује да q мора бити једнако i и отуда нова ознака $(p, i+1)$ представља коректну нову подниску за ту грану ка листу.

Коришћењем парова индекса за представљање ознака гране на свакој грани записују се само два броја. Приметимо да је укупан број грана у стаблу највише $2m-1$, јер је укупан број листова у стаблу m, а сваки унутрашњи чвор повећава број листова бар за један, па унутрашњих чворова



Слика 3.21: Лево стабло представља фрагмент суфиксног стабла ниске $S = abcdefabcuvw$ са експлицитно представљеним ознакама грана. Десно стабло представља исти фрагмент са компресијом ознака грана. Приметимо да се грана са ознаком $2, 3$ може означити такође паром $8, 9$.

има највише $m-1$. Како је број грана највише $2m-1$, то суфиксно стабло користи само $O(m)$ симбола и захтева простор величине $O(m)$.

Размотримо сада однос појединих правила за продужење суфикса са суседним продужењима, односно фазама.

Запажање 1: Правило 3 прекида фазу. Ако се у току фазе за продужење суфикса $S[j..i]$ примени правило 3, исто правило биће примењено и за продужење свих осталих суфикса $S[j+1..i], S[j+2..i], \ldots$ Заиста, ако је примењено правило 3, пут означен са $S[j..i]$ у тренутном стаблу мора да се настави карактером $S[i+1]$. Пошто су $S[j+1..i], S[j+2..i], \ldots$ суфикси ниске $S[j..i]$, они се такође настављају истим карактером $S[i+1]$, па се правило 3 примењује за продужавање свих преосталих индекса.

Приликом примене правила 3 нема потребе да се било шта ради, јер је продужени суфикс већ у стаблу. Поред тога, нова суфиксна веза треба да се дода у стабло само након продужења у којем је примењено правило 2. Ове чињенице и запажање 1 доводе до наредног имплементационог трика.

Трик 2. Када се у фази $i+1$ први пут примени правило 3, фаза је завршена. Другим речима, ако се ово догоди приликом продужења суфикса $S[j..i]$, онда нема потребе експлицитно тражити крај било које ниске $S[k..i], k > j$.

За продужења у фази $i+1$ која су "завршена" након прве примене правила 3 кажемо да су завршена *имплицитно*. Насупрот томе, *експлицитна* продужења су она у току којих се експлицитно тражи крај ниске $S[j..i]$.

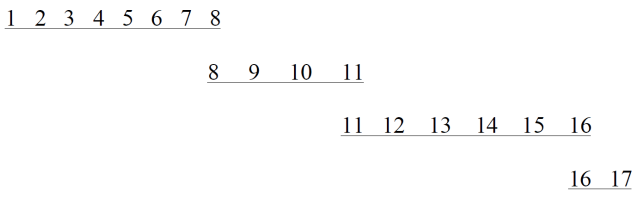
Трик 2 је очигледно добра хеуристика, чији је резултат смањење посла који се обавља. Ипак, још увек није јасно како она утиче на укупну сложеност алгоритма. За то ће нам бити потребно још једно запажање и још један трик.

Запажање 2: Једном лист, увек лист. Ако је у неком тренутку у Укоиеновом алгоритму формиран лист и обележен ознаком j (за суфикс који почиње од j-те позиције у ниски S), тада тај лист остаје лист у свим наредним формираним стаблима у току алгоритма. Ово је тачно јер алгоритам не поседује механизам за продужавање "рачвањем" гране која води ка листу. Прецизније, ако се у неком тренутку појави лист означен са j, у продужењу j у било којој наредној фази увек ће бити примењивано правило 1. Дакле, једном лист, увек лист.

Лист 1 формира се у фази 1, тако да у произвољној фази i постоји почетни низ од првих неколико продужења која се врше применом правила 1 или 2 (јер правило 3 зауставља процес продужавања; прво продужење се увек ради применом правила 1). Нека j_i означава последње продужење у том низу. С обзиром на то да се у фази $i+1$ сва продужења са индексима до j_i раде применом правила 1, односно да било која примена правила 2 формира нови лист, из опажања 2 следи да је $j_i \leq j_{i+1}$. Другим речима, почетни низ продужења где се примењује правило 1, не може бити смањен у наредним фазама. Ово сугерише имплементациони трик да се у фази $i+1$ избегну сва експлицитна продужења од 1 до j_i. Уместо тога биће потребан само константан број корака за извођење имплицитних продужења.

Да бисмо описали овај трик, присетимо се да се ознаке грана у (имплицитним) суфиксним стаблима могу представити паром индекса (p, q) који представљају подниску $S[p..q]$. Присетимо се, такође, да је за сваку грану до листа у T_i, индекс q једнак i, и у фази $i+1$ индекс q се инкрементира на $i+1$, региструјући тако додавање карактера $S[i+1]$ на крај сваког суфикса.

Трик 3. У фази $i+1$, када се грана ка листу први пут формира и када би се означила ознаком $S[p..i+1]$, уместо писања пара индекса $(p, i+1)$ на грани, написати (p, e), где је e симбол који означава тренутни *крај*. Симбол e је глобални индекс који се поставља на $i+1$ по једном у свакој фази. У фази $i+1$, с обзиром да алгоритам зна да ће се правило 1 применити макар у продужењима од 1 до j_i, није потребно радити додатан експлицитан посао којим се имплементира тих j_i продужења. Уместо тога, он само инкрементира променљиву e, а затим спроводи само експлицитна продужења почевши од продужења $j_i + 1$.



Слика 3.22: Приказ једног могућег извршења Укоиеновог алгоритма. Свака линија представља фазу у алгоритму и сваки број представља експлицитно продужење извршено у алгоритму: приказане су четири фазе и седамнаест експлицитних продужења. У произвољним двема суседним фазама налази се највише један индекс где се исто експлицитно продужење извршава у обема фазама.

Линеарност Укоиеновог алгоритма. Са триковима 2 и 3, експлицитна продужења у фази $i+1$ потребна су само у продужењима почевши од j_i+1 до појаве првог продужења у којем се примењује правило 3 или до краја фазе. Сва остала продужења се раде имплицитно. Резимирајући све што је речено, реализација фазе $i+1$ може се описати наредним алгоритмом:

- Инкрементирати индекс e, тако да је његова нова вредност $i+1$ (на основу трика 3, ово коректно реализује сва имплицитна продужења од 1 до j_i).
- Експлицитно извршити сва наредна продужења почевши од j_i+1 до појаве продужења j^* где се примењује правило 3 или док се не заврше сва продужења у тренутној фази (на основу трика 2, ово коректно реализује сва додатна имплицитна продужења од j^*+1 до $i+1$).
- Поставити вредност j_{i+1} на j^*-1, да би се припремила наредна фаза.

Последњи корак алгоритма коректно поставља j_{i+1} због тога што почетни низ продужења где се примењују правила 1 или 2 мора да се заврши непосредно пре прве примене правила 3.

Кључна одлика алгоритма је да фаза $i+1$ почиње са експлицитним продужавањима од продужења j^*, где је j^* последње експлицитно продужење у фази $i+1$. Због тога, две суседне фазе деле највише један индекс (j^*) за који се врши експлицитно продужење (видети слику 3.22). Штавише, фаза $i+1$ се завршава знајући где се завршава ниска $S[j^*..i]$, тако да поновљено продужење j^* у фази $i+2$ може да изврши примену правила за продужење за j^* без икаквог враћања уз стабло, праћења суфиксних веза и прескакања чворова. То значи да је сложеност првог експлицитног продужења у свакој фази једнака $O(1)$. Сада је једноставно доказати главни резултат, формулисан у облику наредне теореме.

Теорема 3.6 (Линеарност Укоиеновог алгоритма)**.** *Коришћењем суфиксних веза и имплементационих трикова 1, 2 и 3, Укоиенов алгоритам формира имплицитна суфиксна стабла T_1 до T_m у $O(m)$ корака.*

Формирање правог суфиксног стабла. Последње формирано имплицитно суфиксно стабло T_m може се претворити у право суфиксно стабло у времену $O(m)$. После додавања терминирајућег карактера \$ на крај ниске S и Укоиенов алгоритам извршава још једну, допунску фазу. Смисао додавања терминирајућег карактера је да ниједан суфикс не може бити префикс неког другог суфикса, па се Укоиенов алгоритам завршава имплицитним суфиксним стаблом у којем се сваки суфикс завршава у листу и тиме је експлицитно представљен. Једина друга измена која је потребна јесте замена променљиве e вредношћу m. Ово се постиже обиласком стабла у времену $O(m)$, посећујући сваку грану ка листу. Резултат који се добија после ових измена је право суфиксно стабло. Тиме је доказана наредна теорема:

Теорема 3.7 (Линеарност формирања суфиксног стабла Укоиеновим алгоритмом)**.** *Сложеност формирања суфиксног стабла ниске S дужине m, заједно са формирањем свих суфиксних веза, је $O(m)$.*

3.3.3 Примене суфиксних стабала

Суфиксна стабла имају бројне примене. Као што смо већ напоменули, једна од примена је за тражење речи у тексту. Ако је задат текст T и реч P, могуће је пронаћи све појаве речи P у T за време $O(n+k)$, где је n дужина речи P, а k је број појава унутар T.

Тачно тражење свих задатих речи

За дате речи $P_1, P_2, \ldots, P_k$ укупне дужине n, суфиксно стабло може да се искористи за проналажење свих појава ових речи у тексту T дужине m алгоритмом сложености $O(m+n+z)$, где је z укупан број појава свих ових речи у T. Суфиксно стабло користи се за предобраду текста T за време $O(m)$, а после тога се тражење свих појава речи (речи се траже једна по једна, независно) завршава за време $O(n+z)$.

Најдужа заједничка подниска две ниске S_1 и S_2

Постоји велики број проблема који обрађују већи број ниски, као што је тражење речи у тексту, аутоматско допуњавање текста, најдужи заједнички подниз две ниске, најдужа палиндромска ниска, ... У овим операцијама све ниске које се користе треба да буду индексиране због брже претраге. Један начин да се то уради је посредством суфиксних стабала. Суфиксно стабло формирано за скуп ниски назива се *уопштено суфиксно стабло*. Једноставан начин за конструкцију уопштеног суфиксног стабла две дате ниске је формирањем нове ниске $S_1\#S_2\$$, која се добија додавањем два различита завршна карактера, и формирањем обичног суфиксног стабла за ову ниску. Сваки лист суфиксног стабла или почиње суфиксом ниске S_1, или S_2, или обе ниске. Ознака сваке гране у уопштеном суфиксном стаблу треба да одговара подниски само једне улазне ниске, те ако постоје ознаке пута које укључују подниске обе улазне ниске, можемо да задржимо само почетни део који припада једној ниски. На основу ознака листова можемо видети да ли је лист суфикс прве или друге ниске (на основу тога да ли ознака листа има вредност из скупа $\{1..|S_1|\}$ или вредност из скупа $\{|S_1|+2..|S_1|+|S_2|+1\}$. На основу овога означићемо унутрашње чворове са S_1, S_2 или S_1S_2 на основу тога да ли су сви листови који одговарају подстаблу са кореном у том унутрашњем чвору суфикси ниске S_1, S_2 или постоји бар један лист који одговара суфиксу ниске S_1 и бар један лист који одговара суфиксу ниске S_2. Ознаке пута до унутрашњег чвора са ознаком S_1S_2 чине подниске обе ниске S_1 и S_2. Ознака пута до чвора коме у стаблу одговара најдужа ниска означеног са S_1S_2 даје најдужи заједнички подниз ове две ниске. Сложеност свих операција које је потребно извршити да се пронађе најдужа заједничка подниска ниски S_1 и S_2 је $O(|S_1|+|S_2|)$.

Глава 4

Сортирање линеарне сложености

Подсетимо се да се сваки алгоритам за сортирање заснован на упоређивањима може моделирати стаблом одлучивања, због чега је његова сложеност $\Omega(n \log n)$. Ова чињеница следи из тога да је временска сложеност алгоритма дефинисаног стаблом одлучивања у најгорем случају једнака висини овог стабла, а с обзиром на то да произвољно стабло одлучивања за сортирање n елемената има бар $n!$ листова (по један за сваку могућу пермутацију елемената на улазу), његова висина је најмање $\log_2(n!) = \Omega(n \log n)$.

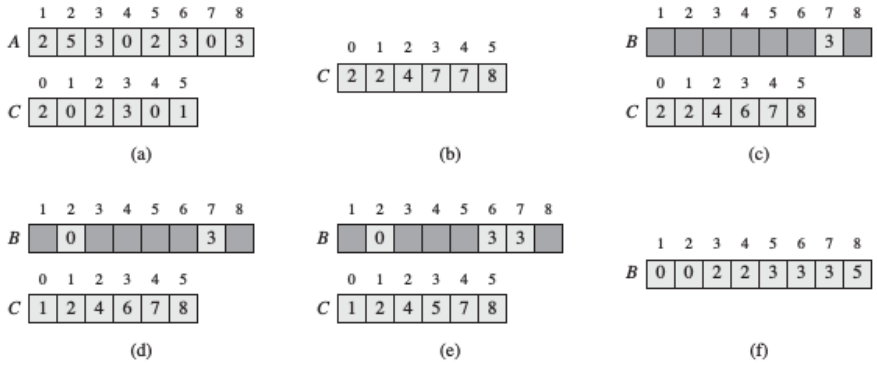
Сортирање је ипак могуће извршавати брже коришћењем специјалних особина бројева, или изводећи алгебарске манипулације са њима. Ово не противречи доказаној доњој граници, јер овакви алгоритми сортирања не користе упоређивања већ рецимо чињеницу да вредности бројева могу да се ефикасно користе као адресе.

4.1 Сортирање пребројавањем

Претпоставимо да низ A садржи n целобројних вредности од којих је свака из опсега $[0, k]$, за неки природни број k. За почетак претпоставимо да су сви елементи низа A међусобно различити. Идеја алгоритма **сортирања пребројавањем** (енг. counting sort) је да се за сваки елемент x низа A одреди број елемената низа A који су мањи од x. Ова информација омогућава постављање елемента x директно на своју позицију у сортираном редоследу. На пример, ако је 17 елемената низа A мање од x, онда је позиција елемента x у сортираном редоследу 18 (претпостављамо да вредности индекса у низу крећу од 1). Да бисмо могли да обрадимо случај када више елемената има исту вредност, потребно је да мало модификујемо предложену шему.

Поред улазног низа $A[1..n]$ у оквиру алгоритма сортирања пребројавањем користе се још два низа: низ $B[1..n]$ који на крају извршавања алгоритма садржи сортирани низ елемената и помоћни низ бројача $C[0..k]$ (после проласка кроз алгоритам $C[i]$ једнако је броју појављивања броја i у низу A).

С обзиром на то да елементи низа A не морају бити различити, декрементирамо вредност $C[A[i]]$ сваки пут када сместимо вредност $A[i]$ у низ B. Тиме се



Слика 4.1: Пример сортирања пребројавањем низа $A[1..8]$, при чему су сви елементи низа A ненегативни цели бројеви не већи од 5. а) Низ A и низ C након постављања елемената низа C тако да $C[i]$ садржи број елемената једнаких i. b) Низ C након извршавања петље којом се вредност $C[i]$ поставља на број елемената мањих или једнаких i. c)-e) Низови B и C након извршавања редом $1, 2$ и 3 итерације петље. f) Сортирани низ B.

Алгоритам Sortiranje_prebrojavanjem(A, n);
Улаз: природни број k и низ $A[1..n]$ целих бројева такав да је $0 \leq A[i] \leq k$ за $1 \leq i \leq n$.
Излаз: B (сортирани низ).
begin
Нека је $C[0..k]$ нови низ
for $i := 0$ **to** k **do**
$C[i] := 0$
for $i := 1$ **to** n **do**
$C[A[i]] := C[A[i]] + 1$
{$C[i]$ сада садржи број елемената једнаких i}
for $i := 1$ **to** k **do**
$C[i] := C[i] + C[i-1]$
{$C[i]$ сада садржи број елемената мањи или једнак од i}
for $i := n$ **downto** 1
$B[C[A[i]]] := A[i]$
$C[A[i]] := C[A[i]] - 1$
end

Слика 4.2: Сортирање пребројавањем.

постиже да уколико постоји још неки елемент са истом вредношћу $A[j]$, један од таквих елемената иде на позицију непосредно пре $A[j]$.

Важно својство алгоритма сортирања пребројавањем је његова *стабилност*: бројеви са истом вредношћу се на излазу појављују у оном редоследу у којем су били на улазу. Ово се постиже кретањем од последње ка првој вредности за индекс у последњој петљи у алгоритму. Кретање у супротном редоследу не би чувало стабилност. Јасно је, ово својство је важно једино

ако се уз вредности које сортирамо чувају и неки додатни (сателитски) подаци. Уколико се не чувају никакви додатни подаци, могли бисмо само проћи кроз низ C добијен након прве петље алгоритма и за сваку вредност $C[i] > 0$ ископирати $C[i]$ копија елемента i у излазни низ.

Укупна сложеност алгоритма је $\Theta(k+n)$. У случају када је $k = O(n)$ сложености је линеарна, $\Theta(n)$. Међутим, уколико је $k = O(n^2)$, овај алгоритам био би квадратне сложености и више би се исплатило применити неки од алгоритама заснованих на поређењу елемената.

4.2 Сортирање разврставањем и сортирање вишеструким разврставањем

Најједноставнији поступак сортирања низа природних бројева би се састојао у томе да се обезбеди довољан број локација, и да се онда сваки елемент смести на своју локацију. Тај поступак зове се **сортирање разврставањем** (енг. bucket sort). Ако се, на пример, сортирају писма према одредиштима, онда је довољно обезбедити једну преграду за свако одредиште, и сортирање је врло ефикасно. Али ако писма треба сортирати према петоцифреном поштанском броју, онда овај метод захтева око 100000 преграда, што поступак чини непрактичним. Према томе, сортирање разврставањем ради добро ако су елементи из малог, једноставног опсега, који је унапред познат. Прелазимо на детаљнији опис овог алгоритма.

Нека је дато n различитих елемената, који су цели бројеви из опсега од 1 до $m \geq n$. Резервише се m локација, а онда се за свако i број x_i ставља на локацију x_i која одговара његовој вредности. После тога се прегледају све локације и из њих се редом покупе елементи. Сложеност овог једноставног алгоритма је дакле $O(m+n)$. Ако је $m = O(n)$, добијамо алгоритам за сортирање линерне сложености. С друге стране, ако је m велико у односу на n (као у случају поштанских бројева), онда је и $O(m)$ такође велико. Поред тога, алгоритам захтева меморију величине $O(m)$, што је још већи проблем за велике m.

Природно уопштење ове идеје је **сортирање вишеструким разврставањем** (енгл. radix sort). Размотримо још једном пример са поштанским бројевима. Сортирање разврставањем у овом случају није погодно, јер је превелики опсег могућих поштанских бројева. Како смањити потребан опсег? Применићемо индукцију по опсегу на следећи начин. Најпре користимо 10 преграда и сортирамо писма према првој цифри поштанског броја. Свака преграда сада покрива 10000 различитих поштанских бројева (одређених са преостале четири цифре поштанског броја). Број операција за ову етапу је $O(n)$. На крају прве етапе имамо 10 преграда, од којих свака одговара мањем опсегу. Даље се проблем за сваку преграду решава рекурзивно. Пошто се опсег после сваке етапе смањује за фактор 10 и пошто поштански бројеви имају пет цифара, довољно је пет етапа. Кад се садржаји преграда сортирају, лако их је објединити у сортирану (уређену) листу. Размотрена верзија сортирања вишеструким разврставањем (цифре се пролазе слева удесно) позната је као сортирање обратним вишеструким разврставањем.

Напоменимо да се опсег може поделити на било који одговарајући начин. У примеру са поштанским бројевима подела је извршена у складу са декадним приказом поштанских бројева. Ако су елементи стрингови које треба сортирати

лексикографски, можемо их упоређивати знак по знак, па се добија лексикографско сортирање.

Размотримо сада другу варијанту исте идеје. Рекурзивна реализација сортирања обратним вишеструким разврставањем захтева помоћне локације (око 50 преграда у примеру са поштанским бројевима; сваки ниво рекурзије има своје преграде). Други начин реализације сортирања вишеструким разврставањема заснива се на примени индукције обрнутим редоследом: сортирање се ради здесна улево, полазећи од најнижих уместо од највиших цифара. Претпостављамо да су елементи велики бројеви, представљени са k цифара у систему са основом d (цифре су из опсега од 0 до $d-1$). Индуктивна хипотеза је врло природна.

Индуктивна хипотеза. Умемо да сортирамо бројеве са мање од k цифара.

Разлика између овог метода и сортирања обратним вишеструким разврставањем је у начину на који се проширује хипотеза (идеја примене индуктивне хипотезе обрнутим редоследом слична је као код Хорнерове шеме). Код датих бројева са k цифара ми најпре игноришемо *највишу* (прву) цифру и сортирамо бројеве према остатку цифара индукцијом. Тако добијамо листу бројева сортираних према најнижих $k-1$ цифара. Затим пролазимо још једном кроз све елементе, и разврставамо их према највишој цифри у d преграда. Коначно, обједињујемо редом садржаје преграда. Овај алгоритам зове се сортирање директним вишеструким разврставањем. Показаћемо да су елементи на крају сортирани по свих k цифара.

Тврдимо да су два елемента сврстана у различите преграде у исправном поретку. За то нам није потребна индуктивна хипотеза, јер је највиша цифра првог од њих већа од највише цифре другог. С друге стране, ако два елемента имају исте највише цифре, онда су они према индуктивној хипотези доведени у исправан редослед пре последњег корака. Битно је да елементи стављени у исту преграду остају у истом редоследу као и пре разврставања. Ово се може постићи употребом листе за сваку преграду и обједињавањем d листа на крају сваке етапе у једну глобалну листу од свих елемената (сортираних према i најнижих цифара). Алгоритам је приказан на слици 4.3.

Сложеност. Потребно је n корака за копирање елемената у глобалну листу GL и d корака за иницијализацију листа $Q[i]$. У главној петљи алгоритма, која се извршава k пута, сваки елеменат се вади из глобалне и ставља у неку од листа $Q[i]$. На крају се све листе $Q[i]$ поново обједињавају у GL. Укупна временска сложеност алгоритма је $O(k(n+d))$.

На пример, уколико је дат низ од n елемената чије су вредности из опсега од 0 до n^2-1, можемо искористити сортирање директним вишеструким разврставањем – свака вредност би била разматрана као двоцифрени број, где је свака цифра из опсега $[0, n-1]$. Овај алгоритам био би сложености $O(2(n+n)) = O(n)$.

4.3 Сортирање просечне линеарне сложености

Размотрићемо сада другу варијанту алгоритма. Приказаћемо алгоритам сортирања разврставањем, при чему се претпоставља да елементи улазног низа $A[1..n]$ имају равномерну расподелу на полуотвореном интервалу $[0, 1)$.

Алгоритам DVR_sort(X, n, k);
Улаз: X (низ од n целих ненегативних бројева са по k цифара).
Излаз: X (сортирани низ).
begin
Претпостављамо да су на почетку сви елементи у глобалној листи GL
{GL се користи због једноставности; листа се може реализовати у X}
for $i := 0$ **to** $d - 1$ **do**
{d је број могућих цифара; $d = 10$ у декадном случају}
иницијализовати листу $Q[i]$ као празну листу;
for $i := k$ **downto** 1 **do**
while GL није празна **do** {разврставање по i-тој цифри}
скини x из GL; {x је први елемент са листе}
$c := i$–та цифра x; {гледано слева удесно}
убаци x у $Q[c]$;
for $t := 0$ **to** $d - 1$ **do** {обједињавање локалних листа}
укључи $Q[t]$ у GL; {додавање на крај листе}
for $i := 1$ **to** n **do** {преписивање елемената из GL у низ x}
скини $X[i]$ из GL
end

Слика 4.3: Сортирање директним вишеструким разврставањем.

Прецизније, претпоставка је да је сваки елемент улазног низа генерисан независно од осталих елемената насумичним одабиром елемента из интервала $[0, 1)$.

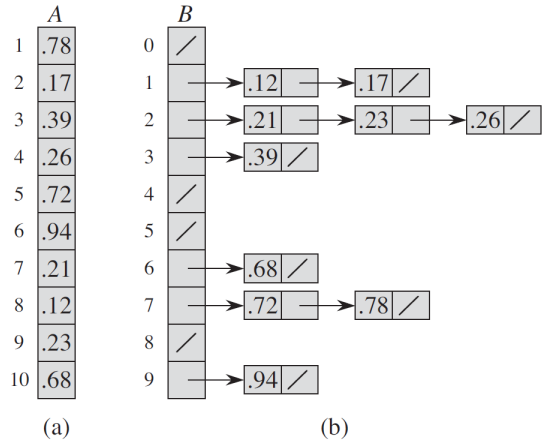
Сортирање разврставањем разбија интервал $[0, 1)$ на тачно n једнаких подинтервала ("прегради"), а затим n улазних бројева распоређује по преградама. Како елементи улазног низа имају равномерну расподелу на интервалу $[0, 1)$, не очекујемо да много бројева упадне у исту прегради. Да бисмо добили излаз, односно сортирани низ елемената полазног низа, сортирамо бројеве у свакој прегради (једноставним алгоритмом, као што је на пример сортирање уметањем), а затим пролазимо кроз прегради, листајући бројеве у свакој од њих.

Алгоритам за сортирање разврставањем претпоставља да је дат низ A дужине n и да за сваки елемент низа A важи: $0 \leq A[i] < 1, 1 \leq i \leq n$. Алгоритам захтева додатни низ B величине n који садржи показиваче на повезане листе (прегради); претпоставља се да постоји механизам за одржавање таквих листи. Алгоритам је приказан на слици 4.4. Пример извршавања овог алгоритма приказан је на слици 4.5.

Да бисмо се уверили у коректност алгоритма, посматрајмо два елемента $A[i]$ и $A[j]$. Претпоставимо без губитка на општости да је $A[i] \leq A[j]$. С обзиром да важи $\lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor$, онда је или елемент $A[i]$ смештен у исту прегради као и $A[j]$, или је смештен у прегради са мањим индексом. Ако се та два елемента сместе у исту прегради, онда их последња петља у алгоритму премешта у исправан редослед. Ако се та два елемента сместе у различите прегради, онда их последња наредба у алгоритму смешта у исправан редослед. Дакле, предложени алгоритам ради коректно.

Алгоритам Sortiranje_razvrstavanjem(A, n);
Улаз: A (низ од n бројева при чему сваки елемент $A[i]$ задовољава $0 \leq A[i] < 1$).
Излаз: X (сортирани низ).
begin
Нека је B иницијално празни низ величине n
for $i := 0$ **to** $n - 1$ **do**
иницијализовати листу $B[i]$ као празну листу;
for $i := 1$ **to** n **do**
убацити $A[i]$ у листу $B[\lfloor n \cdot A[i] \rfloor]$
for $i := 0$ **to** $n - 1$ **do**
сортирати листу $B[i]$ алгоритмом сортирање уметањем
надовезати листе $B[0], B[1], \ldots, B[n-1]$
end

Слика 4.4: Сортирање разврставањем.



Слика 4.5: Пример примене алгоритма *Sortiranje_razvrstavanjem* за $n = 10$. (а) Улазни низ A. (б) Додатни низ B сортираних листи (преграда) након извршења последње петље у алгоритму. Преграда i садржи вредности из полуотвореног интервала $[i/10, (i+1)/10)$. Сортирани излаз добија се надовезивањем редом садржаја свих преграда.

Сложеност. Да бисмо извршили анализу временске сложености алгоритма, приметимо да сви кораци у алгоритму осим корака када се садржај преграда сортира уметањем имају сложеност $O(n)$ у најгорем случају. Дакле, да бисмо одредили време извршавања алгоритма у најгорем случају, потребно је да одредимо укупно време извршавања n позива сортирања уметањем садржаја преграда.

Означимо са n_i случајну променљиву која означава број елемената у прегради $B[i]$. С обзиром на то да је временска сложеност алгоритма сортирања уметањем $O(n^2)$ за низ величине n, једначина која описује

сложеност сортирања разврставањем гласи

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

Просечно трајање сортирања разврставањем можемо одредити израчунавањем очекиване вредности трајања алгоритма, разматрајући очекивање на датој расподели улазних елемената. Примењујући оператор очекивања на претходну једнакост и коришћењем својства линеарности математичког очекивања, добијамо

$$\begin{aligned} E[T(n)] &= E[\Theta(n) + \textstyle\sum_{i=0}^{n-1} O(n_i^2)] \\ &= \Theta(n) + \textstyle\sum_{i=0}^{n-1} E[O(n_i^2)] \\ &= \Theta(n) + \textstyle\sum_{i=0}^{n-1} O(E[n_i^2]). \end{aligned}$$

Покажимо даље да је

$$E[n_i^2] = 2 - 1/n, \tag{4.1}$$

за свако $i = 0, 1, \ldots, n-1$. Није изнанађујуће што свака преграда i има исту вредност $E[n_i^2]$ с обзиром на то да свака вредност у низу A има једнаку вероватноћу да упадне у било коју преграду.

Да бисмо доказали ову једнакост, дефинишемо индикаторске случајне променљиве:

$$X_{ij} = I\{A[j] \text{ упада у преграду } i\} = \begin{cases} 1, & \text{ако } A[j] \text{ упада у преграду } i \\ 0, & \text{ако } A[j] \text{ не упада у преграду } i \end{cases}$$

за свако $i = 0, 1, \ldots, n-1$ и $j = 1, 2, \ldots, n$. Преко ових променљивих могу се изразити бројеви n_i:

$$n_i = \sum_{j=1}^{n} X_{ij}$$

Квадрирањем и прегруписавањем елемената, на основу линеарности очекивања добијамо:

$$\begin{aligned} E[n_i^2] &= E\Big[\Big(\sum_{j=1}^{n} X_{ij}\Big)^2\Big] \\ &= E\Big[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij} \cdot X_{ik}\Big] \\ &= E\Big[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij} \cdot X_{ik}\Big] \\ &= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} E[X_{ij} \cdot X_{ik}] \end{aligned} \tag{4.2}$$

Оценимо посебно сваку од две добијене суме. Индикаторска случајна променљива X_{ij} има вредност 1 са вероватноћом $1/n$, а 0 иначе, те стога важи:

$$E[X_{ij}^2] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n} \tag{4.3}$$

Када је $k \neq j$, променљиве X_{ij} и X_{ik} су независне и стога важи:

$$E[X_{ij} \cdot X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2} \tag{4.4}$$

Заменом ове две вредности у једначину 4.2, добијамо:

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\ &= 1 + \frac{n-1}{n} \\ &= 2 - \frac{1}{n} \end{aligned}$$

чиме је доказана једнакост 4.1.

Коришћењем добијеног очекивања из једначине 4.1, закључујемо да је временска сложеност алгоритма сортирања разврставањем у просеку једнака:

$$T(n) = \Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$$

Јасно је да се алгоритам може применити на произвољни интервал $[a, b)$ уколико су елементи низа са равномерном расподелом на овом интервалу. Чак иако елементи улазног низа немају равномерну расподелу, сортирање разврставањем и даље може да се изврши са линеарно време. Све док улазни низ испуњава услов да је сума квадрата величина преграда линеарна у односу на укупни број елемената, једначина 4.1 нам говори да ће сортирање разврставањем у просеку бити линеарне временске сложености.

Општије, произвољних n реалних бројева могу се сортирати овим алгоритмом за време $O(n)$ ако су добијени независно и из једне исте расподеле вероватноћа. То се постиже на следећи начин: кумулативна функција расподеле вероватноћа $F(x) = P\{X \leq x\}$ пресликава дати скуп реалних бројева на интервал $[0, 1)$, који се дели на n једнаких интервала. Та подела се директно преноси на одговарајуће реалне интервале који имају једнаке вероватноће. Дакле вредности $F(x_1), F(x_2), \ldots, F(x_n)$ распоређујемо редом по интервалима $[i/n, (i+1)/n), i = 0, \ldots, n-1$, а с обзиром да је функција F инвертибилна, можемо на основу вредности функције да реконструишемо њене аргументе. Алтернативно, можемо бројеве $x_1, x_2, \ldots, x_n$ распоредити по интервалима $[F^{-1}(i/n), F^{-1}((i+1)/n)), i = 0, \ldots, n-1$ и очекивани број елемената у сваком од ових подинтервала је 1.

Глава 5

Пробабилистички алгоритми

Алгоритми које смо до сада разматрали били су детерминистички — сваки наредни корак је унапред одређен. Кад се детерминистички алгоритам извршава два пута са истим улазом, начин извршавања је оба пута исти, као и добијени излази. Пробабилистички алгоритми су другачији. Они садрже кораке који зависе не само од улаза, него и од неких случајних догађаја. Постоји много варијација пробабилистичких алгоритама. Овде ћемо размотрити две од њих.

5.1 Одређивање броја из горње половине

Претпоставимо да је дат скуп бројева $x_1, x_2, \ldots, x_n$ и да међу њима треба изабрати неки број из "горње половине", односно број који је већи или једнак од бар $n/2$ осталих. На пример, потребно је изабрати "доброг" студента, при чему је критеријум просечна оцена. Једна могућност је узети највећи број (који је увек у горњој половини). Већ смо видели да је за одређивање максимума потребно $n-1$ упоређивања. Друга могућност је започети са извршавањем алгоритма за налажење максимума, и зауставити се кад се прође половина бројева. Број који је већи или једнак од једне половине бројева је сигурно у горњој половини. Алгоритам захтева око $n/2$ упоређивања. Може ли се овај посао обавити ефикасније? Није тешко показати да је немогуће гарантовати да број припада горњој половини ако је извршено мање од $n/2$ упоређивања. Према томе, описани алгоритам је оптималан.

Овај алгоритам је, међутим, оптималан само ако инсистирамо на *гаранцији*. У много случајева гаранција није неопходна, довољна је пристојна вероватноћа да је решење тачно. На пример, код хеш табела није могуће гарантовати да до колизија неће доћи, али постоји начин за решавање проблема изазваних појавом колизија (операције са хеш табелама се такође могу сматрати пробабилистичким алгоритмима). Ако одустанемо од гаранције, онда постоји бољи алгоритам за налажење елемента из горње половине. Изаберимо на случајан начин два броја x_i и x_j из скупа, тако да је $i \neq j$. Претпоставимо да је $x_i \geq x_j$. Вероватноћа да случајно изабрани број из скупа припада горњој половини је бар $1/2$ (она ће бити већа од $1/2$ ако је више бројева једнако медијани). Вероватноћа да ни један од

бројева x_i, x_j не припада горњој половини је највише 1/4. Због $x_i \geq x_j$ ова вероватноћа једнака је вероватноћи да x_i не припада горњој половини. Према томе, вероватноћа да x_i припада горњој половини је бар 3/4.

Вероватноћа 3/4 да је добијени резултат тачан обично није довољна. Међутим, описани приступ може се уопштити. На случајан начин бирамо k бројева из скупа и одређујемо највећи од њих. На исти начин као у специјалном случају, закључујемо да највећи од k елемената припада горњој половини са вероватноћом најмање $1-2^{-k}$ (он не припада горњој половини акко горњој половини не припада ни један од изабраних бројева, што је догађај са вероватноћом највише 2^{-k}). На пример, ако је $k = 10$, вероватноћа успеха је приближно 0.999, а за $k = 20$ та вероватноћа је око 0.999999. Ако је пак $k = 100$, онда је вероватноћа грешке за све практичне сврхе занемарљива. Имамо дакле алгоритам који бира број из горње половине са произвољно великом вероватноћом, а који извршава мали број упоређивања независно од величине улаза. При томе се претпоставља да се случајни избор једног броја може извршити за константно време; генерисање случајних бројева биће размотрено у одељку 5.2.

За овакав алгоритам обично се каже да је **Монте Карло** алгоритам. Нетачан резултат може се добити са јако малом вероватноћом, али је време извршавања овог Монте Карло алгоритма боље него за најбољи детерминистички алгоритам. Други тип пробабилистичког алгоритма је онај који никад не даје погрешан резултат, али му време извршавања није гарантовано. Овакав алгоритам се може извршити брзо, али се може извршавати и произвољно дуго. Овај тип алгоритма, који са обично зове **Лас Вегас**, користан је ако му је *очекивано* време извршавања мало. У одељку 5.3 биће размотрен Лас Вегас алгоритам који решава један проблем бојења елемената скупа.

Идеја пробабилистичких алгоритама тесно је повезана са извођењем доказа. Коришћење вероватноће за доказивање комбинаторних тврђења је моћна техника. У основи, ако се докаже да неки објекат из скупа објеката има вероватноћу већу од нуле, онда је то индиректан доказ да постоји објекат са тим особинама. Ова идеја може се искористити за конструкцију пробабилистичког алгоритма. Претпоставимо да тражимо објекат са неким особинама, а знамо да ако генеришемо случајни објекат, он ће задовољавати жељени услов са вероватноћом већом од нуле (што је пробабилистички доказ да тражени објекат постоји). Ми затим пратимо пробабилистички доказ, генеришући случајне догађаје кад је потребно, и на крају налазимо жељени објекат се неком позитивном вероватноћом. Овај поступак може се понављати више пута, све до успешног проналажења жељеног објекта.

5.2 Случајни бројеви

Битан елемент пробабилистичких алгоритама је генерисање случајних бројева. Потребно је имати ефикасне методе за решавање овог проблема. Детерминистичка процедура генерише бројеве на фиксирани начин, па тако добијени бројеви не могу бити *случајни* у правом смислу те речи. Ти бројеви су међусобно повезани на сасвим одређени начин. На срећу, то у пракси није велики проблем: довољно је користити тзв. **псеудослучајне бројеве**. Ти бројеви генеришу се детерминистичком процедуром (па дакле нису прави случајни

бројеви), али процедура генерисања је довољно сложена, да друге апликације не "осећају" међузависности између тих бројева.

Овде нећемо детаљније разматрати добијање случајних бројева. Један од начина за добијање псеудослучајних бројева је **линеарни конгруентни метод**. Први корак је избор целог броја X_0 као првог члана низа, случајног броја изабраног на неки независан начин (на пример, текуће време у микросекундама). Остали бројеви израчунавају се на основу диференцне једначине $X_n = aX_{n-1} + b \pmod{m}$, где су a, b и m константе, које се морају пажљиво изабрати. И поред пажљивог избора, овакви низови нису довољно квалитетни ("случајни"). Алтернатива су диференцне једначине вишег реда (са зависношћу од више претходних чланова). На овај начин добија се низ бројева из опсега од 0 до $m-1$. Ако су потребни случајни бројеви из опсега од 0 до 1, онда се чланови овог низа могу поделити са m.

5.3 Један проблем са бојењем

Проблем. Нека је S скуп од n елемената, и нека је $S_1, S_2, \ldots S_k$ колекција сачињена од k његових различитих подскупова, од којих сваки садржи тачно r елемената, $r \geq 2$, при чему је $k \leq 2^{r-2}$. Обојити сваки елемент скупа S једном од две боје, црвеном или плавом, тако да сваки подскуп S_i садржи бар један плави и бар један црвени елемент.

Бојење које задовољава тај услов звaћемо *исправним бојењем*. Испоставља се да под наведеним условима исправно бојење увек постоји. Једноставан пробабилистички алгоритам добија се преправком пробабилистичког доказа постојања таквог бојења:

> *Обојити сваки елемент S случајно изабраном бојом, плавом или црвеном, независно од бојења осталих елемената.*

Јасно је да овај алгоритам не даје увек исправно бојење. Израчунаћемо вероватноћу неуспеха. Вероватноћа да су сви елементи S_i обојени црвено је 2^{-r}, а вероватноћа да су сви обојени истом бојом (црвеном или плавом) је $2 \cdot 2^{-r} = 2^{1-r}$. Случајни догађај (подскуп скупа свих 2^n случајних бојења — елементарних догађаја) A: "неки од скупова S_i је неисправно обојен" је унија свих случајних догађаја A_i: "скуп S_i је неисправно обојен", за $i = 1, 2, \ldots, k$, па важи неједнакост

$$P(A) \leq \sum_{i=1}^{k} P(A_i) = \sum_{i=1}^{k} 2^{1-r} = k2^{1-r} \leq 2^{r-2}2^{1-r} = \frac{1}{2}.$$

Тиме је доказано да исправно бојење постоји (у противном би вероватноћа неисправног бојења била тачно 1). Поред тога, види се да је овај пробабилистички алгоритам добар. Исправност задатог бојења лако се проверава: проверавају се елементи сваког подскупа док се не пронађу два различито обојена елемента. Вероватноћа успешног бојења $p = 1 - P(A)$ је бар $1/2$. Ако се у једном покушају не добије неисправно бојење, поступак (бојење) се понавља. Очекивани број покушаја бојења је мањи или једнак од два. Заиста, вероватноћа да се исправно бојење пронађе у j-том покушају је

$(1-p)^{j-1}p$, па је математичко очекивање броја покушаја

$$\sum_{j=1}^{\infty} j(1-p)^{j-1}p = p\sum_{j=1}^{\infty}\Big((1-p)^j\Big)' = -p(p^{-1})' = \frac{1}{p} \leq 2.$$

Описани алгоритам бојења је очигледно Лас Вегас алгоритам, јер се бојења проверавају једно за другим, а са тражењем се завршава кад се наиђе на исправно бојење. Не постоји гаранција успешног бојења у било ком фиксираном броју покушаја, али је овај алгоритам у пракси ипак врло ефикасан.

Глава 6

Графови

У овом поглављу размотрићемо неколико нових алгоритама над графовима. Графовски алгоритми о којима ће бити речи баве се проблемом конструкције оптималног упаривања у графу, проблемом конструкције оптималног тока у транспортним мрежама и одређивањем Хамилтонових циклуса за одређене класе графова.

6.1 Упаривање

За задати неусмерени граф $G = (V, E)$ **упаривање** је скуп дисјунктних грана (грана без заједничких чворова). Ово име потиче од чињенице да се гране могу схватити као парови чворова. Битно је да сваки чвор припада највише једној грани — то је моногамно упаривање. Чвор који није суседан ни једној грани из упаривања зове се **неупарени** чвор; каже се такође да чвор не припада упаривању. **Савршено упаривање** је упаривање у коме су сви чворови упарени. **Оптимално упаривање** је упаривање са максималним бројем грана. **Максимално упаривање** је пак упаривање које се не може проширити додавањем нове гране. Проблеми који се своде на упаривање појављују се у многим ситуацијама, не само социјалним. Могу се упаривати радници са радним местима, машине са деловима, групе студената са учионицама, итд.

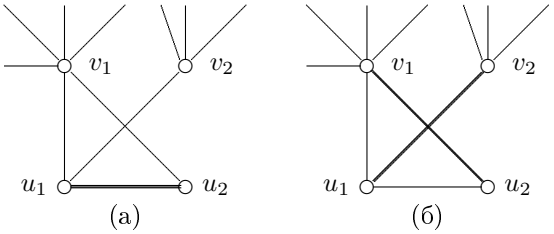
Проблем налажења оптималног упаривања за произвољан граф је тежак проблем. У овом одељку ограничићемо се на два специјална случаја. Први од њих није тако важан — ради се о упаривању у врло густим графовима. Међутим, решење тог проблема илуструје интересантан приступ, који се може уопштити да би се дошло до решења проблема упаривања за бипартитне графове.

6.1.1 Савршено упаривање у врло густим графовима

Нека је $G = (V, E)$ неусмерени граф код кога је $|V| = 2n$ и степен сваког чвора је бар n. Приказаћемо алгоритам за налажење савршеног упаривања у оваквим графовима. Последица овог алгоритма је да ако граф задовољава наведене услове, онда у њему увек постоји савршено упаривање. Користићемо индукцију по величини m упаривања. Базни случај

$m = 1$ решава се формирањем упаривања величине један од произвољне гране графа. Показаћемо да се произвољно упаривање које није савршено може проширити или додавањем једне гране или заменом једне гране двема новим гранама. У оба случаја повећава се величина упаривања за један.

Посматрајмо упаривање M са m грана у графу G, при чему је $m < n$. Најпре проверавамо све гране ван упаривања M да установимо да ли се нека од њих може додати у M. Ако пронађемо такву грану, проблем је решен — нађено је веће упаривање. У противном, M је максимално упаривање. Ако M није савршено упаривање, постоје бар два неупарена чвора v_1 и v_2. Из та два чвора по претпоставци излази најмање $2n$ грана. Све те гране воде ка упареним чворовима (у противном би се у упаривање могла додати нова грана, супротно претпоставци да је оно максимално). Пошто у упаривању M има мање од n грана, а из v_1 и v_2 излази бар $2n$ грана, у упаривању M постоји грана (u_1, u_2) која је суседна са ("покрива") бар три гране из v_1 и v_2. Претпоставимо, без смањења општости, да су то гране (u_1, v_1), (u_1, v_2) и (u_2, v_1), видети слику 6.1(а). Лако је видети да се уклањањем гране (u_1, u_2) из M, и додавањем двеју нових грана (u_1, v_2) и (u_2, v_1) добија веће упаривање, слика 6.1(б).



Слика 6.1: Проширивање упаривања.

Описани алгоритам је пример *похлепног* приступа. У сваком кораку проширивања упаривања за једну грану размотрају се четири чвора и неколико грана које их повезују. У овој ситуацији је то било довољно; међутим, у општем случају је налажење доброг упаривања тежи проблем. Укључивање једне гране у упаривање утиче на избор других грана чак и у удаљеним деловима графа. Показаћемо сада како се овај приступ може применити на други специјални случај проблема упаривања.

6.1.2 Бипартитно упаривање

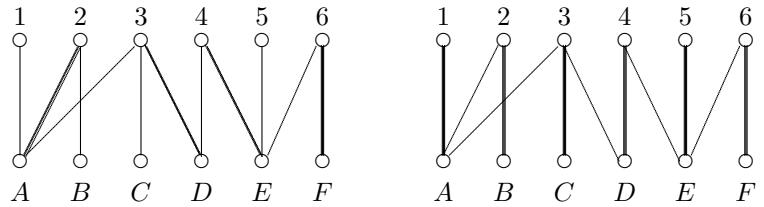
Бипартитни граф је граф чији се чворови могу поделити на два дисјунктна подскупа тако да у графу постоје само гране између чворова из различитих подскупова.

Нека је $G = (V, E, U)$ бипартитни граф у коме су V и U дисјунктни скупови чворова, а E је скуп грана које повезују неке чворове из V са неким чворовима из U.

Проблем. Пронаћи упаривање са максималним бројем грана у бипартитном графу G.

Један од начина да се формулише проблем је следећи: V је скуп девојака, U је скуп младића, а E је скуп "потенцијалних" парова. Циљ је под овим условима оформити што већи број парова младића и девојака.

Директан приступ је формирати парове у складу са неком стратегијом, до тренутка кад даља упаривања више нису могућа — у нади да ће нам стратегија обезбедити оптимално или решење блиско оптималном. Може се, на пример, покушати са похлепним приступом, упарујући најпре чворове малог степена, у нади да ће преостали чворови и у каснијим фазама имати неупарене партнере. Другим речима, најпре упарујемо стидљиве особе, оне са мање познанстава, а о осталима бринемо касније. Уместо да се бавимо анализама оваквих стратегија (што није једноставан проблем), покушаћемо са приступом коришћеним код претходног проблема. Претпоставимо да се полази од максималног упаривања, које не мора бити оптимално. Можемо ли га некако поправити? Погледајмо пример на слици 6.2(а), на коме је упаривање приказано подебљаним гранама. Јасно је да се упаривање може повећати заменом гране $2A$ са две гране $1A$ и $2B$. Ово је трансформација слична оној коју смо применили у претходном проблему. Међутим, не морамо се ограничити заменама једне гране двема гранама. Ако пронађемо сличну ситуацију у којој се неких k грана могу заменити са $k+1$ грана, добијамо алгоритам већих могућности. На пример, упаривање се може даље повећати заменом грана $3D$ и $4E$ са три гране $3C$, $4D$ и $5E$, слика 6.2(б).



Слика 6.2: Проширивање бипартитног упаривања.

Размотримо детаљније ове трансформације. Циљ је повећати број упарених чворова. Полазимо од неупареног чвора v и покушавамо да га упаримо. Пошто полазимо од максималног упаривања, сви суседи чвора v су већ упарени; због тога смо принуђени да из упаривања уклонимо неку од грана које "покривају" суседе v. Претпоставимо да смо изабрали чвор u, суседан са v, који је претходно био упарен са чвором w, на пример. Раскидамо упаривање u са w, и упарујемо v са u. Сада преостаје да пронађемо пара за чвор w. Ако је w повезан граном са неким неупареним чвором, онда смо постигли циљ; такав је био први од горњих случајева. Ако то није случај, онда настављамо даље са раскидањем парова и формирањем нових парова. Да бисмо на основу ове идеје конструисали алгоритам, потребно је да урадимо две ствари: да обезбедимо да се процедура увек завршава, и да покажемо да ако је побољшање могуће, онда ће га процедура сигурно пронаћи. Најпре ћемо формализовати наведену идеју.

Алтернирајући пут P за дато упаривање M је пут од неупареног чвора $v \in V$ до неупареног чвора $u \in U$, при чему су гране пута P наизменично у $E \setminus M$, односно M. Другим речима, прва грана (v, w) пута P не припада M (јер је v неупарен), друга грана (w, x) припада M, и

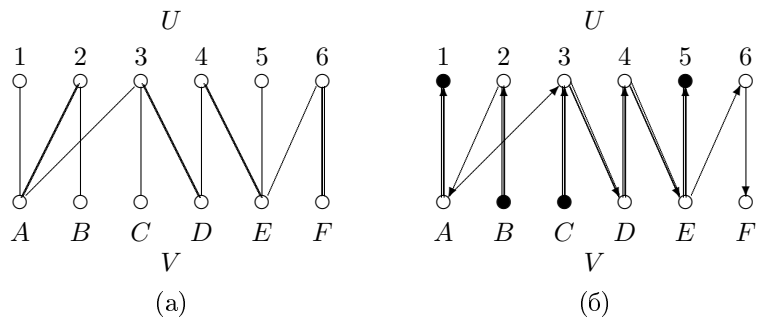
тако даље до последње гране (z, u) пута P која не припада M. Запазимо да су управо алтернирајући путеви у горњим примерима омогућавали повећавање упаривања. Специјално, ако је пут дужине један, онда је то грана која повезује два неупарена чвора; такве гране не постоје у односу на максимално упаривање. Број грана на путу P мора бити непаран, јер P полази из V и завршава у U. Поред тога, међу гранама пута P грана у $E \backslash M$ има за једну више од грана у M. Према томе, ако из упаривања избацимо све гране P које су у M, а укључимо све гране P које су у $E \setminus M$, добићемо ново упаривање са једном граном више. На пример, први алтернирајући пут коришћен за повећање упаривања на слици 6.2(а)) је пут $(1A, A2, 2B)$, и он омогућује замену гране $A2$ гранама $1A$ и $2B$; други алтернирајући пут $(C3, 3D, D4, 4E, E5)$ омогућује замену грана $3D$ и $4E$ гранама $C3$, $D4$ и $E5$.

Јасно је да ако за дато упаривање M постоји алтернирајући пут, онда M није оптимално упаривање. Испоставља се да је тачно и обрнуто тврђење.

Теорема 7 (Теорема о алтернирајућем путу)**.** *Упаривање је оптимално ако и само у односу на њега не постоји алтернирајући пут.*

Доказ ће бити дат као последица општијег тврђења у следећем одељку.

Теорема о алтернирајућем путу директно сугерише алгоритам, јер произвољно упаривање које није оптимално има алтернирајући пут, а алтернирајући пут даје повећано упаривање. Започињемо са похлепним алгоритмом, додајући гране у упаривање све док је то могуће. Онда прелазимо на тражење алтернирајућих путева и повећавање упаривања, све до тренутка кад више нема алтернирајућих путева у односу на последње упаривање. Добијено упаривање је тада оптимално. Пошто алтернирајући пут повећава упаривање за једну грану, а у упаривању има највише $n/2$ грана (где је n број чворова), број итерација је највише $n/2$. Преостаје још један проблем — како проналазити алтернирајуће путеве? Проблем се може решити на следећи начин. Трансформишемо неусмерени граф G у усмерени граф G' усмеравајући гране из M од U ка V, а гране из $E \backslash M$ од V ка U. Слика 6.3(а) приказује полазно максимално упаривање за граф са слике 6.2(а), а слика 6.3(б) приказује одговарајући усмерени граф G'. Алтернирајући пут у G тада одговара усмереном путу од неупареног чвора у V до неупареног чвора у U у графу G'. Такав усмерени пут може се пронаћи било којим поступком обиласка графа, нпр. помоћу DFS. Сложеност обиласка (претраге) је $O(|V|+|E|)$, па је сложеност алгоритма $O(|V|(|V|+|E|))$.



Слика 6.3: Налажење алтернирајућих путева

Побољшање

Пошто комплетан обилазак графа у најгорем случају може да траје колико и налажење једног пута, може се покушати са налажењем више алтернирајућих путева једном претрагом. Потребно је, међутим, да будемо сигурни да су ови путеви независни, односно да њихови скупови чворова буду дисјунктни. Ако су путеви дисјунктни, онда утичу на упаривање различитих чворова, па се могу истовремено искористити. Нови, побољшани алгоритам за налажење алтернирајућих путева је следећи. Најпре примењујемо BFS на граф G' од скупа неупарених чворова у V, слој по слој, до слоја у коме су пронађени неупарени чворови из U. Затим из графа индукованог претрагом у ширину вадимо *максимални* скуп дисјунктних путева у G', којима одговарају алтернирајући путеви у G. То се изводи проналажењем првог пута, уклањањем његових чворова, проналажењем наредног пута, уклањањем његових чворова, итд. (резултат није *оптимални*, него само максимални скуп). Бирамо максимални скуп, да бисмо после претраге добили што веће упаривање; сваки нови дисјунктни пут повећава упаривање за једну грану. На крају повећавамо упаривање коришћењем пронађеног скупа дисјунктних путева. Процес се наставља све док је могуће пронаћи алтернирајуће путеве, односно док је у графу G' неки неупарени чвор из V достижан из неког неупареног чвора из U.

Сложеност. Испоставља се да је у побољшаном алгоритму број итерација $O(\sqrt{|V|})$ у најгорем случају (Хопкрофт и Карп 1973, Hopcroft, Karp; ово тврђење дајемо без доказа). Укупна временска сложеност алгоритма је дакле $O((|V|+|E|)\sqrt{|V|})$.

6.2 Оптимизација транспортне мреже

Проблем оптимизације транспортне мреже је један од основних проблема у теорији графова и комбинаторној оптимизацији. Интензивно је проучаван више од 40 година, па су за њега развијени многи алгоритми и структуре података. Проблем има много варијанти и уопштења. Основна варијанта мреже може се формулисати на следећи начин. Нека је $G = (V, E)$ усмерени граф са два посебно издвојена чвора: s (извор), са улазним степеном 0, и t (понор) са излазним степеном 0. Свакој грани $e \in E$ придружена је позитивна тежина $c(e)$, **капацитет** гране e. Капацитет гране је мера тока који може бити пропуштен кроз грану. За овакав граф кажемо да је **транспортна мрежа** (или једноставније мрежа). **Ток** је функција f дефинисана на E која задовољава следеће услове:

(1) $0 \leq f(e) \leq c(e)$: ток кроз произвољну грану не може да премаши њен капацитет;

(2) за све чворове $v \in V \setminus \{s, t\}$ је $\sum_u f(u, v) = \sum_w f(v, w)$: укупан ток који улази у произвољни чвор v различит од s, t једнак је укупном току који излази из њега ("нестишљивост", закон очувања, односно конзервације тока).

Ова два услова имају за последицу да је укупан ток који излази из s једнак укупном току који улази у t. У то се можемо уверити на следећи начин. Увешћемо најпре појам **пресека**. Нека је A произвољан подскуп V

такав да садржи s, а не садржи t. Означимо са $B = V \setminus A$ скуп преосталих чворова. Пресек одређен скупом A је скуп грана $(v, u) \in E$ таквих да $v \in A$ и $u \in B$. Интуитивно, пресек је скуп грана које раздвајају s од t. Индукцијом по броју чворова у A лако се показује да укупан ток кроз пресек не зависи од A. Специјално, за $A = \{s\}$ пресек обухвата гране које излазе из s, а за $A = V \setminus \{t\}$ пресек чине гране које улазе у t. Према томе, укупан ток који излази из s једнак је укупном току који улази у t. Проблем који нас занима је максимизирање тока. Један начин да се описани проблем схвати као реалан физички проблем, је да замислимо да мрежу чине цеви за воду. Свака цев има свој капацитет, а услови које ток треба да задовољи су природни. Циљ је "протерати" кроз мрежу што већу количину воде у јединици времена.

Показаћемо најпре да се проблем бипартитног упаривања може свести на проблем оптимизације транспортне мреже. То може на први поглед да изгледа бескорисно, пошто већ знамо да решимо проблем бипартитног упаривања, а не знамо решење проблема оптимизације транспортне мреже — редукција је дакле у погрешном смеру. Међутим, за решавање проблема оптимизације транспортне мреже може се применити поступак сличан поступку за налажење оптималног бипартитног упаривања.

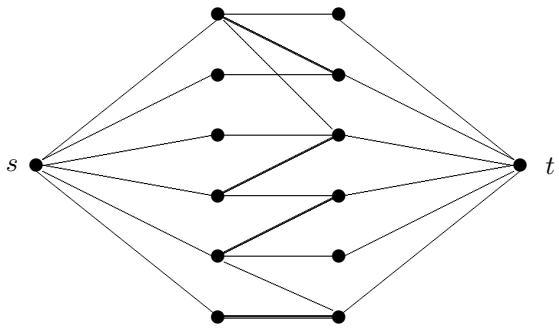
Задатом бипартитном графу $G = (V, E, U)$ у коме треба пронаћи упаривање са највећим могућим бројем грана (оптимално упаривање) додајемо два нова чвора s и t, повезујемо s гранама са свим чворовима из V, а све чорове из U повезујемо са t. Означимо добијени граф са G' (видети слику 6.4, на којој су све гране усмерене слева удесно). Пошто свим гранама доделимо капацитет 1, добијамо регуларан проблем оптимизације транспортне мреже на графу G'. Нека је M неко упаривање у G. Упаривању M може се на природан начин придружити ток у G'. Додељујемо ток 1 свим гранама из M и свим гранама које s или t повезују са упареним чворовима. Свим осталим гранама додељујемо ток 0. Укупан ток једнак је тада броју грана у упаривању M. Може се показати да је M оптимално упаривање ако и само ако је одговарајући целобројни ток у G' оптималан. У једном смеру доказ је једноставан: ако је ток оптималан и одговара упаривању, онда се не може наћи веће упаривање, јер би му одговарао већи ток. Да бисмо извели доказ у другом смеру, потребно је да некако применимо идеју алтернирајућих путева на ток у транспортној мрежи, и да покажемо да ако нема алтернирајућих путева, онда је одговарајући ток оптималан.

Повећавајући пут у односу на задати ток f је усмерени пут од s до t, који се састоји од грана из G, не обавезно у истом смеру; свака од тих грана (v, u) треба да задовољи тачно један од следећа два услова:

(1) (v, u) има исти смер као и у G, и $f(v, u) < c(v, u)$. У том случају грана (v, u) је **директна грана** Директна грана има капацитет већи од тока, па се може повећати ток кроз њу. Разлика $c(v, u) - f(v, u)$ зове се **слек** те гране.

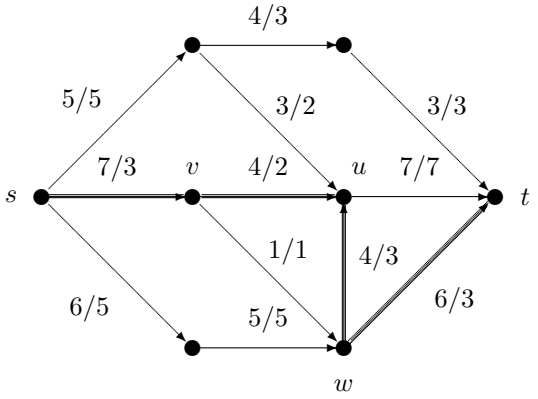
(2) (v, u) има супротан смер у G, и $f(v, u) > 0$. У овом случају грана (v, u) је **повратна грана**. Део тока из повратне гране може се "позајмити".

Повећавајући пут је уопштење алтернирајућег пута, и има исти смисао за транспортне мреже као алтернирајући пут за бипартитно упаривање. Ако постоји повећавајући пут у односу на ток f, онда f није оптимални



Слика 6.4: Свођење бипартитног упаривања на оптимизацију транспортне мреже (све гране усмерене су слева удесно).

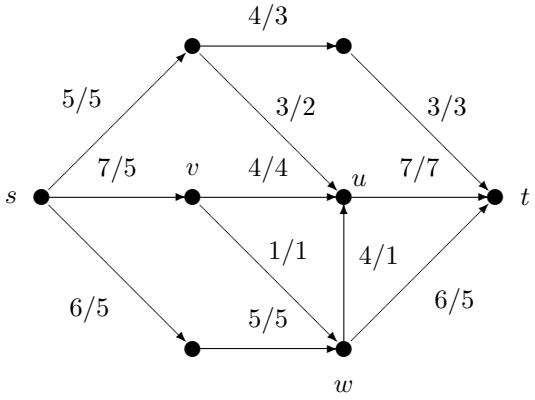
ток. Ток f може се повећати повећавањем тока кроз повећавајући пут на следећи начин. Ако су све гране повећавајућег пута директне гране, онда се кроз њих може повећати ток, тако да сва ограничења и даље остану задовољена. Највеће могуће повећање тока је у овом случају тачно једнако минималном слеку међу гранама пута. Случај повратних грана је нешто сложенији, видети пример на слици 6.5. Свака грана означена је са два броја a/b, при чему је a капацитет, а b тренутни ток. Јасно је да се укупан ток не може директно повећати, јер не постоји пут од s до t који се састоји само од директних грана. Ипак, постоји начин да се укупан ток повећа.



Слика 6.5: Пример мреже са повећавајућим путем.

Пут $s - v - u - w - t$ је повећавајући пут. Допунски ток 2 може се спровести до u од s (2 је минимални слек на директним гранама до u). Ток 2 може се *одузети* (позајмити) од $f(w, u)$. Тиме се постиже задовољење услова (2) (закона очувања тока) за чвор u, јер је u имао повећање тока за 2 из повећавајућег пута, а затим смањење дотока за 2 из повратне гране. У чвору w је сада излазни ток смањен за 2, па га треба повећати кроз неку излазну грану. Са "протеривањем" тока може се наставити на исти начин од w, повећавањем тока кроз директне гране и смањивањем тока кроз повратне гране. У овом случају постоји само још једна директна грана (w, t) која достиже t, и проблем је решен. Пошто само директне гране могу

да излазе из s, односно да улазе у t, укупан ток је повећан. Повећање је једнако мањем од следећа два броја: минималног слека директних грана, односно минималног тока повратних грана. На слици 6.6 приказана је иста мрежа са промењеним током; испоставља се да је нови ток у ствари оптималан.



Слика 6.6: Резултат повећавања тока у мрежи са слике 6.5.

Из реченог следи да ако у мрежи постоји повећавајући пут, онда ток није оптималан. Обрнуто је такође тачно.

Теорема 8 (Теорема о повећавајућем путу)**.** *Ток кроз транспортну мрежу је оптималан ако и само ако у односу на њега не постоји повећавајући пут.*

Доказ. Доказ у једном смеру смо већ видели — ако у мрежи постоји повећавајући пут, онда ток није оптималан. Претпоставимо сада да у односу на ток f не постоји ни један повећавајући пут, и докажимо да је тада f оптимални ток. За произвољан пресек (одређен скупом A, $s \in A$, $t \in B \equiv V \setminus A$) дефинишемо капацитет, као збир капацитета његових грана које воде из неког чвора A у неки чвор B. Јасно је да ни један ток не може бити већи од капацитета произвољног пресека. Заиста, укупан ток из s једнак је збиру токова кроз гране пресека од A ка B, умањеном за ток кроз гране пресека од B ка A, па је мањи или једнак од збира капацитета грана које воде од A ка B, односно од капацитета пресека. Према томе, ако пронађемо ток са вредношћу једнаком капацитету неког пресека, онда је тај ток оптималан. Са доказом настављамо у том правцу: показаћемо да ако у односу на ток не постоји повећавајући пут, онда је укупан ток једнак капацитету неког пресека, па дакле оптималан.

Нека је f ток у односу на који не постоји повећавајући пут. Нека је $A \subset V$ скуп чворова v таквих да у односу на ток f постоји повећавајући пут од s до v (прецизније, постоји пут од s до v такав да на њему за све директне гране e важи $f(e) < c(e)$, а за све повратне гране e' важи $f(e') > 0$). Јасно је да $s \in A$ и $t \notin A$, јер по претпоставци за f не постоји повећавајући пут. Према томе, A дефинише пресек. Тврдимо да за све гране (v, w) тог пресека важи $f(v, w) = c(v, w)$ ако је $v \in A$, $w \in B$ ("директне" гране), односно $f(v, w) = 0$ ако је $v \in B$, $w \in A$ ("повратне гране"). Заиста, у противном би директна грана (v, w) продужавала повећавајући пут до чвора $w \notin A$,

супротно претпоставци да такав пут постоји само до чворова из A. Слично, повратна грана (v, w) продужавала би повећавајући пут до чвора $v \notin A$. Дакле, укупан ток једнак је капацитету пресека одређеног скупом A, па је ток f оптималан.

Доказали смо следећу важну теорему.

Теорема 9 (Теорема о максималном току и минималном пресеку)**.** *Оптимални ток у мрежи једнак је минималном капацитету пресека.*

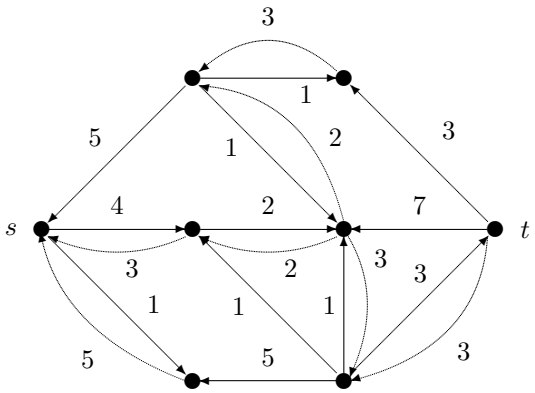
Теорема о повећавајућем путу има за последицу и следећу теорему.

Теорема 10 (Теорема о целобројном току)**.** *Ако су капацитети свих грана у мрежи целобројни, онда постоји оптимални ток са целобројном вредношћу.*

Доказ. Тврђење је последица теореме о повећавајућем путу. Сваки алгоритам који користи само повећавајуће путеве доводи до целобројног тока ако су сви капацитети грана целобројни. Ово је очигледно, јер се може кренути од тока 0, а онда се укупан ток после сваке употребе повећавајућег пута повећава за цели број. До истог закључка долази се и на други начин: капацитет сваког пресека је целобројан, па и минималног.

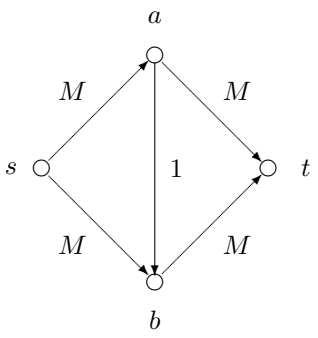
Вратимо се сада на проблем бипартитног упаривања. Јасно је да сваки алтернирајући пут у G одговара повећавајућем путу у G', и обрнуто. Последица теореме о повећавајућем путу је теорема о алтернирајућем путу из претходног одељка. Ако је M оптимално упаривање, онда за њега не постоји алтернирајући пут, па у G' не постоји повећавајући пут, а одговарајући ток је оптималан. С друге стране, постоји оптимални целобројни ток, и он мора да одговара упаривању, јер је сваки чвор у V повезан само једном граном (са капацитетом 1) са s; због тога, укупан ток кроз сваки чвор из V може да буде највише 1. Исто важи и за чворове из скупа U. Ово упаривање мора бити оптимално, јер ако би се могло повећати, онда би постојао већи укупни ток.

Теорема о повећавајућем путу непосредно се трансформише у алгоритам. Полази се од тока 0, траже се повећавајући путеви, и на основу њих повећава се ток, све до тренутка кад повећавајући путеви више не постоје. Тражење повећавајућих путева може се извести на следећи начин. Дефинишемо **резидуални граф** у односу на мрежу $G = (V, E)$ и ток f, као мрежу $R = (V, F)$ са истим чворовима, истим извором и понором, али промењеним скупом грана и њихових тежина. Сваку грану $e = (v, w)$ са током $f(e)$ замењујемо са највише две гране $e' = (v, w)$ (ако је $f(e) < c(e)$; капацитет e' једнак је слеку гране e: $c(e') = c(e) - f(e)$), односно $e'' = (w, v)$ (ако је $f(e) > 0$; капацитет e'' је $c(e'') = f(e)$). Ако се на овај начин добију две паралелне гране, замењују се једном, са капацитетом једнаком збиру капацитета паралелних грана. На слици 6.7 приказан је резидуални граф за мрежу са слике 6.5, у односу на ток задат на тој слици. Гране резидуалног графа одговарају могућим гранама повећавајућег пута. Њихови капацитети одговарају могућем повећању тока кроз те гране. Према томе, повећавајући пут је обичан усмерени пут од s до t у резидуалном графу. Конструкција резидуалног графа захтева $O(|E|)$ корака, јер се свака грана проверава тачно једном.



Слика 6.7: Резидуални граф мреже са слике 6.5 у односу на ток дефинисан на тој слици.

На несрећу, избор повећавајућег пута на произвољан начин може се показати врло неефикасним. Време извршења таквог алгоритма у најгорем случају може да чак и не зависи од величине графа. Посматрајмо мрежу на слици 6.8. Оптимални ток је очигледно $2M$. Међутим, могли бисмо да кренемо од повећавајућег пута $s-a-b-t$ кроз који се ток може повећати само за 1. Затим бисмо могли да изаберемо повећавајући пут $s-b-a-t$ који опет повећава ток само за 1. Процес може да се понови укупно $2M$ пута, где M може бити врло велико, без обзира што граф има само четири чвора и пет грана. Пошто се вредност M може представити са $O(\log M)$ бита, сложеност овог алгоритма је у најгорем случају експоненцијална функција величине улаза (број M је део улаза).



Слика 6.8: Пример мреже на којој тражење повећавајућих путева може бити неограничено неефикасно.

Горе наведена могућност је врло непожељна, али се може избећи. Едмондс и Карп (Edmonds, Carp) су 1972. године показали да ако се међу могућим повећавајућим путевима увек бира онај са најмањим бројем грана, онда је број повећавања највише $(|V|^3 - |V|)/4$. То води алгоритму који је у најгорем случају полиномијалан у односу на величину улаза. Од тог времена предложено је више различитих алгоритама, сложенијих и мање сложених, а више њих достиже горњу границу сложености $O(|V|^3)$ у најгорем случају.

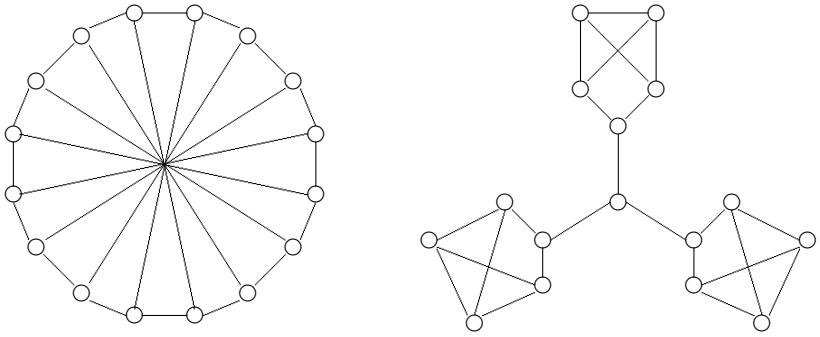
Овде их нећемо описивати.

6.3 Хамилтонови циклуси

Проблем. Задат је граф $G = (V, E)$. Пронаћи у G прости циклус који сваки чвор из V садржи тачно једном.

Такав циклус зове се **Хамилтонов циклус**. Граф који садржи Хамилтонов циклус зове се **Хамилтонов граф**. Проблем има усмерену и неусмерену верзију; ми ћемо се бавити само неусмереном варијантом.

За разлику од проблема Ојлерових циклуса, проблем налажења Хамилтонових циклуса (односно карактеризације Хамилтонових графова) је врло тежак. Да би се проверило да ли је граф Ојлеров, довољно је знати степене његових чворова. За утврђивање да ли је граф Хамилтонов, то није довољно. Заиста, два графа приказана на слици 6.9 имају по 16 чворова степена 3 (дакле имају исте степене чворова), али је први Хамилтонов, а други очигледно није. Овај проблем спада у класу NP-комплетних проблема. У овом одељку приказаћемо једноставан поступак налажења Хамилтоновог циклуса у специјалној класи врло густих графова.



Слика 6.9: Два графа са по 16 чворова степена 3, од којих је први Хамилтонов, а други није.

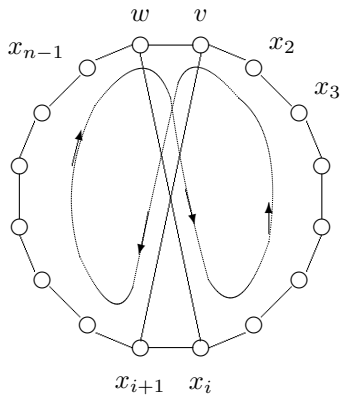
Нека је $G = (V, E)$ повезан неусмерен граф и нека за произвољан чвор $v \in V$ $d(v)$ означава његов степен. Следећи проблем је специјални случај тражења Хамилтоновог циклуса у врло густим графовима. Показаћемо да услови проблема гарантују да је граф Хамилтонов.

Проблем. Дат је повезан неусмерен граф $G = (V, E)$ са $n \geq 3$ чворова, такав да сваки пар несуседних чворова v и w задовољава услов $d(v)+d(w) \geq n$. Пронаћи у G Хамилтонов циклус.

Алгоритам се заснива на индукцији по броју грана које треба уклонити из комплетног графа да би се добио задати граф. База индукције је комплетан граф. Сваки комплетан граф са бар три чвора садржи Хамилтонов циклус, који је лако пронаћи.

Индуктивна хипотеза. Умемо да пронађемо Хамилтонов циклус у графовима који задовољавају наведене услове ако имају бар $n(n-1)/2 - m$ грана.

Сада треба да покажемо како пронаћи Хамилтонов циклус у графу са $n(n-1)/2-(m+1)$ грана који задовољава услове проблема. Нека је $G=(V,E)$ такав граф. Изаберимо произвољна два несуседна чвора v и w у G (то је могуће ако граф није комплетан), и посматрајмо граф G' који се од G добија додавањем гране (v,w). Према индуктивној хипотези ми умемо да пронађемо Хамилтонов циклус у графу G'. Нека је x_1, x_2, ..., x_n, x_1 такав циклус у G' (видети слику 6.10). Ако грана (v,w) није део циклуса, онда је исти циклус део графа G, па је проблем решен. У противном, без смањења општости може се претпоставити да је $v=x_1$ и $w=x_n$. Према датим условима је $d(v)+d(w)\geq n$. Потребно је у графу пронаћи нови Хамилтонов циклус.



Слика 6.10: Модификација Хамилтоновог циклуса после избацивања гране (v,w).

Посматрајмо све гране графа G суседне са чворовима v или w. Тврдимо да под задатим условима постоје два чвора x_i и x_{i+1} таква да у G постоје гране (w,x_i) и (v,x_{i+1}). Да бисмо то доказали, претпоставимо супротно, да ни за једно i, $2\leq i\leq n-2$, не постоје истовремено обе гране (w,x_i) и (v,x_{i+1}). Нека је чвор w везан са чвором x_{n-1} и k чворова x_{i_1}, $x_{i_2},\ldots,x_{i_k}$, при чему је $2\leq i_1<i_2<\cdots<i_k\leq n-2$. Тада по претпоставци чвор v није везан ни са једним од чворова x_{i_1+1}, $x_{i_2+1},\ldots,x_{i_k+1}$, па је $d(v)\leq n-2-k$ (од укупно $n-1$ могућих грана из v не постоје гране ка w, нити ка k наведених чворова, различитих од w). Због тога је $d(w)+d(v)\leq(k+1)+(n-2-k)=n-1$; ово је у контрадикцији са претпоставком да је $d(w)+d(v)\geq n$. Дакле, за неко i постоје гране (w,x_i) и (v,x_{i+1}). Од ових двеју грана може се формирати нови Хамилтонов циклус $v(=x_1),x_{i+1},x_{i+2},\ldots,w(=x_n),x_i,x_{i-1},\ldots,v$, који не садржи грану (v,w), видети слику 6.10.

Реализација. Директна примена овог доказа полази од комплетног графа, из кога се једна за другом избацују гране које не припадају задатом графу. Боље је решење почети са много мањим графом, на следећи начин. У датом графу G најпре проналазимо дугачак пут (на пример, помоћу DFS), а онда додајемо нове гране тако да пут продужимо до Хамилтоновог циклуса. Тако је добијен већи граф G'. Обично је довољно додати само неколико грана. Међутим, чак и у најгорем случају биће додата највише $n-1$ грана. Полазећи од G', доказ теореме се затим примењује итеративно, све док се не пронађе Хамилтонов циклус у G. Укупан број корака за

замену једне гране је $O(n)$. Потребно је заменити највише $n-1$ грана, па је временска сложеност алгоритма $O(n^2)$.

Глава 7

Редукције

7.1 Увод

Започећемо ово поглавље једном анегдотом. Математичару је објашњено како може да скува чај: треба да узме чајник, напуни га водом из славине, стави чајник на штедњак, сачека да вода проври, и на крају стави чај у воду. А онда су га питали: како може да скува чај ако има чајник, пун вреле воде? Једноставно, каже он; просуће воду и тако свести проблем на већ решен.

У овом поглављу позабавићемо се идејом редукције, односно свођења једног проблема на други. Показаћемо да редукције, иако понекад нерационалне, могу бити и врло корисне. На пример, ако убаците у сандуче поште на Новом Београду писмо за човека који станује одмах поред те поште, писмо ће, без обзира на близину одредишта, прећи пут до Главне поште у Београду, па назад до поште на Новом Београду, и тек онда ће га поштар однети вашем познанику. У много случајева није лако препознати специјални случај и за њега скројити ефикасније специјално решење. У пракси је често ефикасније све специјалне случајеве третирати на исти начин. На такву ситуацију редовно се наилази и при конструкцији алгоритама. Кад наиђемо на проблем који се може схватити као специјални случај другог проблема, користимо познато решење. То решење понекад може бити превише опште или превише скупо. Међутим, у многим случајевима је коришћење општег решења најлакши, најбржи и најелегантнији начин да се проблем реши. Овај принцип се често користи. За неке рачунарске проблеме, на пример рад са неком базом података, обично није неопходно написати програм који решава само тај проблем. Опште решење не мора бити најефикасније, али је много једноставније искористити управо њега.

Претпоставимо да је дат проблем P, који изгледа компликовано, али личи на познати проблем Q. Може се независно (од почетка) решавати проблем P, или се може искористити неки од метода за решавање Q и применити на P. Постоји, међутим, и трећа могућност. Може се покушати са налажењем **редукције**, односно свођења једног проблема на други. Неформално, редукција је решавање једног проблема коришћењем "црне кутије" која решава други проблем. Редукцијом се може постићи један од два циља, зависно од смера. Решење P, које користи црну кутију за

решавање Q, може се трансформисати у алгоритам за решавање P, ако се зна алгоритам за решавање Q. С друге стране, ако се за P зна да је тежак проблем, и зна се доња граница сложености за алгоритме који решавају P, онда је то истовремено и доња граница сложености за проблем Q. У првом случају је редукција искоришћена за добијање информације о P, а у другом — о Q.

На пример, у одељку 7.4.2 разматрају се проблеми множења и квадрирања матрица. Јасно је да се квадрирање може извести применом алгоритма за множење; према томе, квадрирање матрице може се свести на множење матрица. У одељку 7.4.2 се показује да је могуће помножити две матрице применом алгоритма за квадрирање; другим речима, множење матрица своди се на квадрирање. Сврха ове друге редукције је извођење доказа да се квадрирање матрице не може извести асимптотски брже од израчунавања производа две произвољне матрице (под условима који су размотрени у одељку 7.4.2).

У овом поглављу видећемо неколико примера редукција. Налажење редукције једног проблема на други може бити корисно чак и ако не даје нову доњу или горњу границу сложености проблема. Редукција омогућује боље разумевање оба проблема. Она се може искористити за налажење нових техника за напад на проблем или његове варијанте. На пример, редукција се може искористити за конструкцију паралелног алгоритма за решавање проблема.

Један од ефикасних начина за употребу редукција је дефинисање врло општег проблема, на кога се могу свести многи други проблеми. Такав проблем треба да буде довољно општи, да покрије широку класу проблема, док са друге стране треба да буде довољно једноставан, да би имао ефикасно решење. Линеарно програмирање, један од таквих проблема, размотрићемо у одељку 7.3.

Поменимо и то да смо се и раније већ сретали са примерима редукција — на пример, са редукцијом проблема налажења транзитивног затворења на проблем налажења свих најкраћих путева.

7.2 Примери редукција

У овом одељку размотрићемо неколико примера употребе редукција као метода за конструкцију ефикасних алгоритама.

7.2.1 Цикличко упоређивање стрингова

Почињемо са једноставном варијантом проблема проналажења узорка у тексту.

Проблем. Нека су $A = a_0a_1 \dots a_{n-1}$ и $B = b_0b_1 \dots b_{n-1}$ два стринга дужине n. Установити да ли је стринг B циклички померај стринга A.

Проблем је установити да ли постоји индекс k, $0 \leq k \leq n-1$, такав да је $a_i = b_{(k+i) \bmod n}$ за све i, $0 \leq i \leq n-1$. Означимо овај проблем са CUS, а основни проблем тражења речи у тексту са TUT. Проблем CUS може се решити, на пример, изменом алгоритма KMP. Постоји бољи начин да се дође до решења. Идеја је формулисати CUS као регуларни пример улаза за

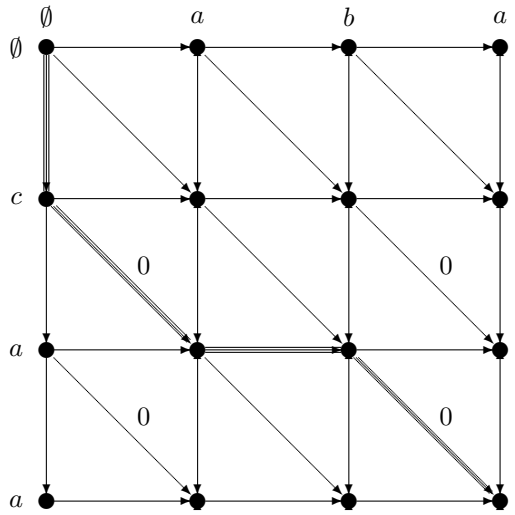
проблем TUT. Другим речима, тражимо неки *текст* T и *реч* P, тако да је проналажење P у T еквивалентно утврђивању да је B циклички померај A. Ако нам то пође за руком, онда се решење TUT са стринговима T и P може искористити за решавање CUS са стринговима A и B. Кад се проблем размотри на овај начин, лако је видети решење: за T треба узети AA (односно стринг A надовезан на самог себе). Јасно је да је B циклички померај A ако и само ако је B подстринг стринга AA. Пошто проблем TUT умемо да решимо за линеарно време, дошли смо до алгоритма за решавање CUS линеарне сложености.

7.2.2 Редукције са упоређивањем низова

Размотримо проблем одређивања едит растојања две ниске: дати су низови знакова $A = a_1a_2 \ldots a_n$, $B = b_1b_2 \ldots b_m$, а задатак је променити A знак по знак, и тако га учинити једнаким низу B. Дозвољене су три едит операције — *уметање*, *брисање* и *замена* знака. Знају се цене сваке операције, а циљ је минимизација цене едитовања. Решење које смо раније разматрали заснива се на попуњавању табеле димензија $n \times m$, у којој сваки елемент одговара *парцијалном* едитовању: елемент табеле у пресеку i-те врсте и j-те колоне је најмања цена трансформисања првих i знакова ниске A у j првих знакова ниске B. Циљ се постиже израчунавањем елемента у доњем десном углу табеле. Видели смо да се сваки елемент може израчунати на основу само три "претходна" елемента, што одговара трима могућим варијантама за последњу едит операцију.

Исти проблем може се посматрати на други начин ако табели придружимо усмерени граф. Сваки елемент табеле одговара чвору графа, односно парцијалном едитовању. Грана (v, w) у графу постоји ако се едитовање које одговара чвору w добија додавањем једне едит операције едитовању које одговара чвору v. Пример таквог графа дат је на слици 7.1, где је $A = caa$ и $B = aba$. Хоризонталне гране одговарају уметањима, вертикалне брисањима, а дијагоналне заменама знакова. Тако, на пример, пут истакнут на слици 7.1 одговара брисању c, замени a са a, уметању b и још једној замени a са a. У основној варијанти проблема цене грана су 1, изузев дијагоналних грана које одговарају замени знака истим знаком, чија је цена 0. Проблем се на тај начин трансформише у обичан проблем налажења свих растојања од задатог чвора у графу. Свакој грани додељена је тежина, и ми тражимо најкраћи пут од чвора $(0, 0)$ до чвора (n, m). Дакле, проблем налажења едит растојања сведен је на проблем налажења најкраћих растојања од задатог чвора у графу.

Налажење свих растојања од задатог чвора у општем случају није лакше од директног решавања проблема (алгоритам за налажење најкраћих растојања од задатог чвора је сложености $O((|E| + |V|) \log |V|)$, где је са $|V|$ означен број чворова који је овде једнак $(m + 1) \cdot (n + 1)$), док је број грана пропорционалан броју чворова (из свих чворова осим оних на доњој и десној ивици крећу по три гране). Ова редукција ипак није бескорисна. Размотримо, на пример, следећу варијанту проблема упоређивања низова. Цене едит операција не морају да зависе само од појединачних знакова. Цена уметања блока знакова унутар другог низа може бити различита од уметања истог броја знакова једног по једног, на различитим местима.



Слика 7.1: Граф који одговара низовима $A = caa$ и $B = aba$.

Исто може да буде случај и са брисањима. Другим речима, уместо да цене додељујемо појединачним уметањима, брисањима и заменама, цене се могу доделити блоковима уметања, брисања и замена, независно од њихове величине. Или другачије, уметању блока од k знакова може се доделити цена $c_1 + c_2k$, где је c_1 "почетна цена", а c_2 цена за сваки наредни знак. Постоји много других корисних метрика. Оне се много лакше моделирају коришћењем формулације у виду проблема најкраћих путева, него прилагођавањем полазног проблема. Исто тако, могу се додати нове гране са погодно изабраним ценама, не мењајући суштину проблема.

7.2.3 Налажење троуглова у неусмереном графу

Постоји тесна веза између графова и матрица. Граф $G = (V, E)$ са n чворова $v_1, v_2, \ldots v_n$, може се представити својом матрицом повезаности — квадратном матрицом $A = (a_{ij})$ реда n, таквом да је $a_{ij} = 1$ ако $(v_i, v_j) \in E$, односно $a_{ij} = 0$ у осталим случајевима. Ако је G неусмерени граф, матрица A је симетрична. Ако је G тежински граф, онда се он такође може представити квадратном матрицом $A = (a_{ij})$ реда n, при чему је елемент a_{ij} једнак тежини гране (v_i, v_j), односно ∞, ако те гране нема у графу. Постоје и други начини да се матрица придружи графу. Тако се графу $G = (V, E)$ са n чворова и m грана може придружити $n \times m$ матрица у којој је (i, j)-ти елемент једнак 1 ако и само ако је i-ти чвор суседан са j-том граном.

Везе графова и матрица не задржавају се само на репрезентацији. Многе особине графова могу се боље разумети анализом одговарајућих матрица. Слично, многе особине матрица откривају се посматрањем одговарајућих графова. Није изненађујуће да се многи алгоритамски проблеми могу решити коришћењем ове аналогије. То ћемо илустровати једним примером.

Проблем. Нека је $G = (V, E)$ неусмерени повезани граф са n чворова и m

грана. Потребно је установити да ли у G постоји **троугао**, односно таква три чвора да између свака два од њих постоји грана.

Директно решење обухвата проверу свих трочланих подскупова скупа чворова. Подскупова има $\binom{n}{3} = n(n-1)(n-2)/6$, а пошто се сваки од њих може проверити за константно време, временска сложеност оваквог алгоритма је $O(n^3)$. Може се конструисати и алгоритам сложености $O(mn)$ (заснован на провери свих парова $(grana, čvor)$ — да ли граде троугао), који је бољи ако граф није густ. Може ли се ово даље побољшати? Приказаћемо сада алгоритам који је асимптотски бржи, и суштински је другачији. Сврха примера је да илуструје везу између графовских и матричних алгоритама.

Нека је A матрица повезаности графа G. Пошто је граф G неусмерен, матрица A је симетрична. Размотримо везу елемената матрице $B = A^2 = AA$ (производ је обичан производ матрица) и графа G. Према дефиницији производа матрица је

$$B[i,j] = \sum_{k=1}^{n} A[i,k] \cdot A[k,j].$$

Из ове једнакости следи да је услов $B[i,j] > 0$ испуњен ако и само ако постоји индекс k, такав да су оба елемента $A[i,k]$ и $A[k,j]$ јединице. Другим речима, $B[i,j] > 0$ је испуњено ако и само ако постоји чвор v_k, такав да је $k \neq i$, $k \neq j$ и да су оба чвора v_i и v_j повезана са v_k (претпостављамо да граф нема петљи, односно $A[i,i] = 0$ за $i = 1, 2, \ldots, n$). Према томе, у графу постоји троугао који садржи темена v_i и v_j ако и само ако је v_i повезан са v_j и $B[i,j] > 0$. Коначно, у G постоји троугао ако и само ако постоје такви индекси i, j да је $A[i,j] = 1$ и $B[i,j] > 0$.

Наведена анализа сугерише алгоритам. Треба израчунати матрицу $B = A^2$ и проверити испуњеност услова $A[i,j] = 1$, $B[i,j] > 0$ за свих n^2 парова (i,j). Сложеност провере овог услова је $O(n^2)$, па преовлађујући део временске сложености алгоритма потиче од множења матрица. Тиме је проблем налажења троугла у графу сведен на проблем множења матрица (прецизније, квадрирања матрице, али као што ћемо видети у одељку 7.4.2, та два проблема су еквивалентна). За множење матрица може се искористити Штрасенов алгоритам и тако добити алгоритам за налажење троугла у графу сложености $O(n^{2.81})$. Прецизније, описана редукција показује да је сложеност овог графовског проблема $O(M(n))$, где је $M(n)$ сложеност множења Булових матрица реда n.

7.3 Редукције на проблем линеарног програмирања

Претходни одељак садржи примере редукција између алгоритама у различитим областима. Покушали смо да трансформишемо један проблем у други, да бисмо могли да искористимо познати алгоритам. У овом одељку такође се разматрају редукције, али је приступ другачији. Уместо да тражимо кандидата за редукцију сваки пут кад наиђемо на нови проблем, разматрамо "супер–проблеме", на које се могу свести многи други проблеми. Један такав супер–проблем (можда најважнији) је **линеарно програмирање**.

7.3.1 Увод и дефиниције

Садржај многих проблема је максимизирање или минимизирање неке функције, која задовољава задате услове. На пример, код оптимизације транспортне мреже циљ је максимизирање фукције тока, под условом да су задовољени услови капацитета грана и конзервације тока. Линеарно програмирање је општа формулација проблема код којих су функција и ограничења линеарни. Нека је $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ вектор *променљивих*. **Циљна функција** је линеарна функција променљивих — компоненти вектора $\mathbf{x}$:

$$c(\mathbf{x}) = \sum_{i=1}^{n} c_i x_i, \tag{7.1}$$

где су c_i константе. Циљ линеарног програмирања је пронаћи вредност $\mathbf{x}$ која задовољава задата ограничења (која ће бити наведена) и *максимизира* циљну функцију. Касније ћемо видети да је лако заменити минимизирање циљне функције њеним максимизирањем. Претходно наводимо општи облик задатка линеарног програмирања са три типа ограничења, при чему се у конкретним проблемима не морају појављивати сва три типа ограничења. Показаћемо да се општи проблем може свести на проблем са само два типа ограничења.

Нека су $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_k$ реални вектори врсте једнаке дужине n, и нека су $b_1, b_2, \ldots, b_k$ реални бројеви. **Неједнакосна ограничења** су ограничења облика

$$\mathbf{a}_j \cdot \mathbf{x} \leq b_j, \quad 1 \leq j \leq k, \tag{7.2}$$

при чему су сви симболи сем компоненти $\mathbf{x}$ константе, а $\cdot$ је ознака за матрични производ. Слична су **једнакосна ограничења**

$$\mathbf{e}_j \cdot \mathbf{x} = d_j, \quad 1 \leq j \leq m, \tag{7.3}$$

при чему су $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_m$ такође вектори врсте дужине n, а $d_1, d_2, \ldots, d_m$ су реални бројеви.

Обично се додају и посебна **ненегативна ограничења** (иако се она могу схватити као специјални случај неједнакосних ограничења)

$$x_j \geq 0 \quad \text{ за све } j \in P, \tag{7.4}$$

где је P задати подскуп скупа $\{1, 2, \ldots, n\}$.

Проблем линеарног програмирања може се формулисати на следећи начин: максимизирати функцију $c(\mathbf{x})$ (7.1), под условом да су задовољена неједнакосна (7.2), једнакосна (7.3) и ненегативна ограничења (7.4). Разуме се да конкретни проблеми не морају да садрже ограничења свих типова.

За овако дефинисан проблем у општем случају постоји више еквивалентних формулација. На пример, једнакосна ограничења типа $\mathbf{e}_j \cdot \mathbf{x} = d_j$ могу се еквивалентно заменити са два неједнакосна ограничења $\mathbf{e}_j \cdot \mathbf{x} \leq d_j$ и $\mathbf{e}_j \cdot \mathbf{x} \geq d_j$. Слично, неједнакосно ограничење $\mathbf{a}_i \cdot \mathbf{x} \leq b_i$ може се заменити са два еквивалентна, $\mathbf{a}_i \cdot \mathbf{x} + y_i = b_i$ и $y_i \geq 0$, при чему је y_i нова променљива. У оба случаја замена једног скупа ограничења другим може да повећа број ограничења.

На овом месту нећемо се бавити алгоритмом за решавање проблема линеарног програмирања. Важно је знати да је овај проблем из класе P,

односно да се за његово решавање зна алгоритам полиномијалне сложености. Напоменимо и то да су постојећи алгоритми за решавање проблема линеарног програмирања у пракси довољно ефикасни, и да свођење неког проблема на линеарно програмирање није само увежбавање редукција, него може да буде и добар начин да се тај проблем реши.

7.3.2 Примери редукције на линеарно програмирање

У пракси су проблеми ретко директно задати као проблеми линеарног програмирања. Обично је неопходно увести одговарајуће дефиниције да би се проблем уклопио у ову формулацију.

Транспортни проблем

Овај проблем разматран је у одељку 6.2. Нека промењљиве $x_1, x_2, \ldots, x_n$ представљају вредности тока за свих n грана $e_1, e_2, \ldots, e_n$, Циљна функција је вредност укупног тока

$$c(\mathbf{x}) = \sum_{j \in S} x_j,$$

где је S скуп грана које излазе из извора s. Неједнакосна ограничења одговарају ограничењима капацитета

$$x_i \leq c_i, \quad 1 \leq i \leq n,$$

где је c_i капацитет гране e_i. Једнакосна ограничења одговарају условима одржања тока

$$\sum_{e_i \text{ излази из } v} x_i = \sum_{e_j \text{ улази у } v} x_j, \qquad \text{за све } v \in V \setminus \{s, t\}.$$

На крају, све промењљиве треба да задовоље ненегативна ограничења $x_i \geq 0$ за $i \in \{1, 2, \ldots, n\}$. Вредности $\mathbf{x}$ које максимизирају цињну функцију уз ова ограничења одговарају очигледно оптималном току.

Добротворни проблем

Претпоставимо да n особа желе да упуте помоћ у k установа. Особа i има ограничење s_i на укупан прилог у току године, као и ограничење a_{ij} на износ прилога установи j (на пример, a_{ij} може да буде 0 за неке установе). У општем случају s_i је мање од $\sum_{j=1}^{k} a_{ij}$, па свака особа треба да донесе одлуку о томе коме да упути колики прилог. Претпоставимо даље да свака установа j има ограничење t_j на укупан износ који може да прими (ово ограничење не мора увек да буде присутно, али је ипак интересантно). Циљ је направити алгоритам који максимизира збир прилога.

Овај проблем је уопштење проблема упаривања из одељка 6.1.2. Проблем се може решити поступком сличним поступцима за налажење оптималног упаривања, а може се формулисати и као проблем линеарног програмирања. Укупно има nk промењљивих x_{ij}, $1 \leq i \leq n$, $1 \leq j \leq k$, где је x_{ij} износ који особа i уплаћује установи j. Циљна функција је

$$c(\mathbf{x}) = \sum_{i,j} x_{ij}.$$

Ограничења су следећа:

$$x_{ij} \leq a_{ij} \quad \text{за све } i, j,$$

$$\sum_{j=1}^{k} x_{ij} \leq s_i \quad \text{за све } i,$$

$$\sum_{i=1}^{n} x_{ij} \leq t_j \quad \text{за све } j.$$

Поред тога, све промењљиве наравно морају бити ненегативне: $x_{ij} \geq 0$ за све i, j.

Проблем придруживања

Променимо мало добротворни проблем тако да свака особа може да уплати прилог само једној установи, и да свака установа може да прими прилог од само једне особе. Тако добијамо стандардни проблем упаривања, али са тежинама. Сваком могућем упаривању придружен је збир прилога, а циљ је не само пронаћи оптимално упаривање, него и максимизирати суму прилога. Ово је тежински проблем бипартитног упаривања, или **проблем придруживања**.

Промењљиве у овом проблему не могу бити исте као у претходном. Потребно је на неки начин окарактерисати упаривање. Треба обезбедити да је са сваким чвором повезана тачно једна изабрана грана. То се може постићи додељивањем по једне промењљиве x_{ij} свакој грани (i, j), тако да је $x_{ij} = 1$ ако је грана изабрана, односно $x_{ij} = 0$ у противном. Циљна функција је

$$c(\mathbf{x}) = \sum_{i,j} a_{ij} x_{ij},$$

а ограничења су

$$\sum_{j=1}^{k} x_{ij} = 1 \quad \text{за све } i,$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \text{за све } j.$$

Поред тога, све промењљиве x_{ij} треба да буду ненегативне. Наведена ограничења обезбеђују да је за сваки чвор изабрана највише једна грана.

Са оваквом формулацијом постоји један озбиљан проблем. Промењљиве треба да представљају избор типа "да"–"не", а њихове оптималне вредности могу да буду реални (дакле не цели) бројеви! Потребно је додати ограничење да промењљиве могу узимати само вредности 0 или 1. У општем случају такав проблем је тешко решити. Линеарни програми чије промењљиве морају бити цели бројеви зову се **целобројни линеарни програми**, а њихово решавање је **целобројно линеарно програмирање**. Међутим, иако се линеарни програми могу ефикасно решавати, целобројни линеарни програми су обично (али не увек) врло тешки. Шта више, важи да је проблем целобројног линеарног програмирања NP-комплетан.

Тражење максималне клике

Клика је подскуп C скупа чворова V графа $G = (V, E)$ такав да за свака два чвора x и y из C важи $(x, y) \in E$. Задатак је одредити максималну клику у датом графу, односно клику са највећим бројем чворова. Показаћемо сада како се проблем клика може формулисати као проблем целобројног линеарног програмирања.

Сваком чвору графа G придружујемо по једну промењиву x_i, тако да је $x_i = 1$ ако чвор v_i припада клики, односно $x_i = 0$ у супротном. Циљна функција је:

$$c(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

и задатак је максимизирати је, јер је потребно укључити у клику што већи број чворова. Постоји по једно ограничење за сваки чвор:

$$0 \leq x_i \leq 1, i = 1, 2, \ldots, n$$

Ограничење које нам обезбеђује да изабрани скуп чворова буде клика еквивалентно је ограничењу да од свака два несуседна чвора највише један чвор може да припада скупу. Ово ограничење кодирамо наредним условом:

$$x_i + x_j \leq 1 \quad \text{за сваки пар чворова } (v_i, v_j) \notin E,\ i < j$$

Додатно ограничење је да за сваки чвор важи да је вредност x_i из скупа $\{0, 1\}$.

Овакав проблем целобројног линеарног програмирања може се решити алгоритмом гранања са одсецањем, коришћењем одговарајућег линеарног програма (који решава исти проблем, али без ограничења целобројности) за израчунавање горњих граница. Решење линеарног програма може се састојати само од целих бројева; у том случају полазни проблем је решен. Међутим, вероватније је да ће у решењу неке промењљиве имати нецелобројне вредности. Претпоставимо, на пример, да је решење линеарног програма придруженог проблему клика $(0.1, 1, \ldots, 0.5)$ и $z = 7.8$. Пошто линеарни програм максимизира циљну функцију са мање ограничења од целобројног линеарног програма, максимум који он пронађе је горња граница за максимум који може да пронађе целобројни линеарни програм. Према томе, не може се очекивати проналажење клике величине веће од 7. Ова информација може бити корисна приликом претраге. Као и код обичне претраге, бирамо неке вредности промењљивих и напредујемо низ стабло, при чему ако је теме b у стаблу син темена a, онда је проблем који одговара темену b потпроблем проблема из темена a, који се добија фиксирањем вредности неке промењљиве на вредност 0 или 1; тиме је потпроблем који одговара произвољном чвору потпроблем полазног проблема. На пример, потпроблем може да одговара прикључивању чворова v, w клики, и елиминисању из ње чворова u, x, тј. покушава се са налажењем највеће клике са v, w, а без u, x. Ако се том приликом добије решење линеарног програма које је *мање* од величине већ пронађене клике, онда чинимо корак назад, и напуштамо ту варијанту. То је суштина метода гранања и одсецања. Покушавамо да нађемо горње границе (или доње, ако се циљна функција минимизира) које

ће омогућити одсецање неперспективних подстабала на што нижем нивоу (што ближе корену стабла).

Резултат линеарног програма може се искористити и при избору редоследа претраге. На пример, ако је $x_2 = 1$ у нецелобројном решењу, може се претпоставити да је $x_2 = 1$ и у целобројном решењу. Та претпоставка не мора бити тачна, али је пример врсте хеуристика које тражимо. Покушавамо да повећамо "вероватноћу" брзог налажења оптималног решења; при томе је јасно да не могу све такве одлуке да буду "исправне", јер је проблем NP-комплетан. Можемо да ставимо $x_2 = 1$, изменимо у складу са тим ограничења (нпр. променимо вредности променљивих за све чворове несуседне са v_2 на 0), и решимо резултујући линеарни програм. Ако у неком тренутку модификовани линеарни програм има максималну вредност $z = a$, при чему је a мање од величине највеће пронађене клике, та грана у стаблу претраге се може напустити.

Према томе, линеарни програм користи се на два начина: за добијање горњих граница и тиме за напуштање неперспективиних подстабала (одсецање), односно за доношења одлука о усмеравању претраге. Очекује се да решавање потпроблема који "највише обећава", учини непотребним решавање великог дела других потпроблема. Учесталост одсецања, односно ефикасност целог алгоритма, зависи од хеуристике за формирање потпроблема и избора наредног потпроблема за испитивање. Хеуристика зависи од конкретног проблема који се решава, и у овој области спроводе се широка истраживања.

Алгоритми са гранањем и одсецањем гарантују проналажење оптималног решења ако се сви потпроблеми истраже или "одсеку". Ако извршавање траје предуго, може се прекинути, и тако добити апроксимација — најбоље решење пронађено до тог тренутка.

Минимални покривач грана

За дати неусмерени граф $G = (V, E)$ покривач грана је подскуп чворова V такав да за сваку грану из E важи да је суседна бар једном од чворова из овог скупа. Разматра се проблем проналажења минималног покривача грана за дати граф G.

Овај проблем се једноставно може преформулисати у проблем целобројног линеарног програмирања. Сваком чвору из V додељујемо по једну променљиву x_i. Циљна функција коју треба минимизовати је

$$c(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

а ограничења су облика

$$x_i + x_j \geq 1 \quad \text{за сваку грану } e_{ij} = (v_i, v_j) \in E$$

Ова ограничења одговарају услову да је бар један од крајева сваке гране укључен у покривач. Додатно, сваки чвор може бити укључен или не у покривач, те додајемо услов $x_i \in \{0, 1\}$ за све $i = 1, 2, \ldots, n$ што нам сугерише да је ово проблем целобројног линеарног програмирања.

Бојење графа

Размотримо проблем бојења чворова датог графа минималним бројем боја тако да никоја два суседна чвора нису обојена истом бојом. С обзиром на то да је максималан потребан број боја једнак n (ако је граф комплетан), уводимо n променљивих $y_k, k = 1, 2, \ldots, n$ којима кодирамо услов да ли се боја k користи ($y_k = 1$) или не ($y_k = 0$). Уводимо и n^2 променљивих x_{ik} које означавају да ли је чвор i обојен бојом k. Циљна функција коју треба минимизирати је:

$$c(\mathbf{x}) = \sum_{k=1}^{n} y_k$$

а скуп услова је:

$$\sum_{k=1}^{n} x_{ik} = 1, \ i = 1, 2, \ldots, n,$$

$$x_{ik} - y_k \leq 0, \ i, k = 1, 2, \ldots, n,$$

$$x_{ik} + x_{jk} \leq 1 \quad \text{за сваку грану } (i, j) \in E \text{ и } k = 1, 2, \ldots, n,$$

$$x_{ik}, y_k \in \{0, 1\}.$$

Прво ограничење одговара услову да се сваки чвор мора обојити тачно једном од n могућих боја. Друго ограничење еквивалентно је услову да се чвор i може обојити бојом k само ако се боја k користи у бојењу. Треће ограничење кодира услов да се било која два суседна чвора не могу обојити истом бојом, а последње ограничење да су у питању целобројне вредности.

7.4 Примена редукција на налажење доњих граница

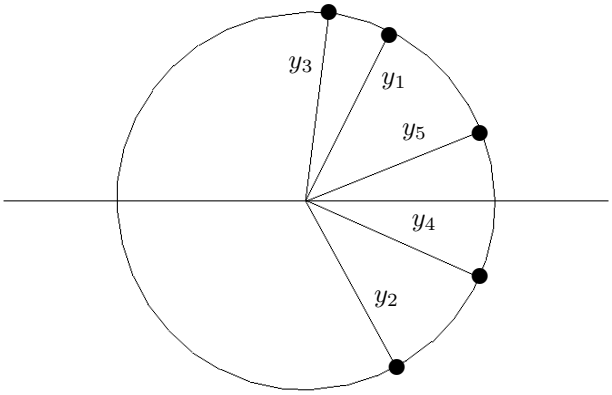
Ако покажемо да се произвољан алгоритам за решавање проблема A може модификовати (не повећавајући му при томе значајно временску сложеност) тако да решава проблем B, онда је доња граница за проблем B истовремено и доња граница за проблем A. Заиста, сваки алгоритам за A је истовремено и алгоритам за B, па је доња граница за B већа или једнака од доње границе за A. Приказаћемо три примера доказа доње границе сложености заснована на редукцији.

7.4.1 Доња граница за конструкцију простог многоугла

Посматрајмо проблем повезивања скупа тачака у равни простим многоуглом (видети одељак 2.3). Видели смо како се тај проблем може решити коришћењем сортирања. Испоставља се да, под одређеним претпоставкама, овај проблем не може да се реши брже од сортирања. Због тога се алгоритам који смо видели за налажење простог многоугла не може побољшати ако се не побољша сортирање (под "побољшањем" се подразумева побољшање за више од константног фактора).

Теорема 11. *Нека је познат алгоритам за конструкцију простог многоугла временске сложености $O(T(n))$. Тада постоји алгоритам за сортирање сложености $O(T(n) + n)$.*

Доказ. Размотримо n тачака на кружници, слика 7.2. Једини начин да се те тачке повежу простим многоуглом је да се свака тачка повеже са својим суседима на кружници. У противном, ако две тачке које су суседи не би биле повезане, дуж која садржи једну од те две тачке раздвајала би преостале тачке на две групе, које се не могу повезати не пресецајући ову дуж. Посматрајмо сада инстанцу $x_1, x_2, \ldots, x_n$ проблема сортирања. Ако бисмо имали алгоритам ("црну кутију") за решавање проблема простог многоугла, бројеве бисмо могли да сортирамо на следећи начин. Улаз $x_1, x_2, \ldots, x_n$ најпре пресликавамо у углове $y_1, y_2, \ldots, y_n$, из опсега од 0 до 2π, са истим међусобним односима као и $x_1, x_2, \ldots, x_n$. То се може постићи линеарном функцијом $y = 2\pi(x_i - x_{min})/(x_{max} - x_{min})$ која интервал (x_{min}, x_{max}) пресликава на интервал $(0, 2\pi)$; овде је (x_{min}, x_{max}) произвољан интервал који садржи све тачке x_i. Углови се затим на природан начин пресликавају у тачке на јединичној кружници: броју x_i одговара тачка на кружници, чији њен полупречник са фиксном полуправом заклапа угао y_i. Ово пресликавање бројева у тачке изводи се за линеарно време. Сада се може искористити црна кутија за повезивање овог скупа тачака простим многоуглом, за време $O(T(n))$. Као што смо већ видели, многоугао мора да спаја сваку тачку са њеним суседима на кружници. Због тога, пролазећи тачке оним редом којим су оне сложене да би чиниле темена многоугла, добијамо низ тачака сортиран по угловима, а тиме и сортиран полазни низ бројева. Сложеност оваквог сортирања је $O(T(n) + n)$.



Слика 7.2: Придруживање тачака на кружници бројевима.

Да бисмо добили доњу границу за проблем налажења простог многоугла, морамо бити пажљиви у погледу подразумеваног модела рачунања. Доња граница $\Omega(n \log n)$ за проблем сортирања доказана је под претпоставком да се ради о моделу стабла одлучивања. Да би ова граница могла да се искористи за проблем налажења простог многоугла, мора се користити исти модел. Другим речима, мора се претпоставити да црна кутија која решава проблем налажења простог многоугла користи $O(T(n))$ упоређивања, на начин који је у складу са моделом стабла одлучивања. Теорема дакле мора да садржи ту претпоставку. Затим се мора показати да је и редукција у складу са моделом стабла одлучивања. У овом случају редукција је регуларна јер приликом доказа доње границе за сортирање није било никаквих

ограничења на тип питања дозвољених у оквиру стабла одлучивања. Дакле, упоређивање које ради са x и y координатама које одговарају углу y_i рачуна се као једно упоређивање у стаблу одлучивања. Стабло одлучивања које решава проблем налажења простог многоугла може се трансформисати у стабло одлучивања за сортирање, без значајне промене висине.

Последица 12. Ако се претпостави модел стабла одлучивања, конструкција простог многоугла који повезује скуп од задатих n тачака у равни захтева у најгорем случају $\Omega(n \log n)$ упоређивања.

Ова редукција установљава чињеницу да је сортирање централни део решавања проблема конструкције простог многоугла.

7.4.2 Једноставне редукције са матрицама

На симетричне матрице (оне којима је елемент (i, j) једнак елементу (j, i) за сваки пар индекса (i, j)) често се наилази у пракси. Природно је упитати се да ли је могуће на једноставнији начин множити симетричне матрице него произвољне матрице. Није нелогично да симетрија омогућује проналажење бољих израза за множење нпр. матрица реда 3. То би могло да произведе асимптотски бољи алгоритам за множење симетричних матрица. Показаћемо да то ипак није могуће, односно да је множење две симетричне матрице, са тачношћу до на константни фактор, исте сложености као и множење две произвољне матрице.

Означимо проблем израчунавања производа две произвољне матрице са PrM, а проблем израчунавања производа две симетричне матрице са SimM. Јасно је да проблем SimM није *тежи* од PrM, јер је SimM специјални случај PrM. Претпоставимо сада да имамо алгоритам који решава SimM. Показаћемо да се тај алгоритам може искористити као црна кутија за решавање општијег проблема PrM. Нека су A и B две произвољне квадратне матрице реда n. Означимо са A^T транспоновану матрицу матрице A. Непосредно се проверава да је тачан следећи израз у коме се појављује производ две *симетричне* $2n \times 2n$ матрице

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} 0 & B^T \\ B & 0 \end{pmatrix} = \begin{pmatrix} AB & 0 \\ 0 & A^T B^T \end{pmatrix}. \tag{7.5}$$

Овде 0 означава $n \times n$ матрицу чији су сви елементи нуле. Редукција је последица чињенице да су матрице са леве стране симетричне. Њихов производ може се израчунати коришћењем алгоритма за решавање проблема SimM. Међутим, горњи леви блок овог производа је управо производ AB. Према томе, проблем PrM може се решити применом алгоритма за решавање SimM на матрице двоструко веће димензије. Тако долазимо до следећег тврђења.

Теорема 13. *Ако постоји алгоритам за множење две реалне симетричне $n \times n$ матрице за време $O(T(n))$, при чему је $T(2n) = O(T(n))$, онда постоји алгоритам за множење две произвољне реалне $n \times n$ матрице за време $O(T(n) + n^2)$.*

Доказ. Ако су дате две произвољне $n \times n$ матрице A и B, њихов производ налазимо коришћењем алгоритма за множење симетричних матрица на

основу једнакости (7.5). Потребно је извршити $O(n^2)$ операција за израчунавање A^T и B^T и формирање две симетричне матрице, и $T(2n)$ операција да се помноже две добијене симетричне матрице, из чега непосредно следи тврђење теореме.

Претпоставка да је $T(2n) = O(T(n))$ није прејака, јер је, на пример, свака полиномијална функција задовољава. Наведена редукција је интересантна само у оквиру доказа доње границе. Теорема тврди да је немогуће искористити симетрију приликом множења матрица, и тако добити асимптотски бржи алгоритам. Наводимо још једну сличну редукцију.

Теорема 14. *Ако постоји алгоритам који израчунава квадрат реалне $n \times n$ матрице за време $O(T(n))$, при чему је $T(2n) = O(T(n))$, онда постоји алгоритам за множење две произвољне реалне $n \times n$ матрице за време $O(T(n) + n^2)$.*

Доказ. Као и у доказу теореме 13, треба пронаћи матрицу чији квадрат садржи довољно информација за израчунавање производа две задате матрице. Решење се заснива на следећем изразу:

$$\begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}^2 = \begin{pmatrix} AB & 0 \\ 0 & BA \end{pmatrix}.$$

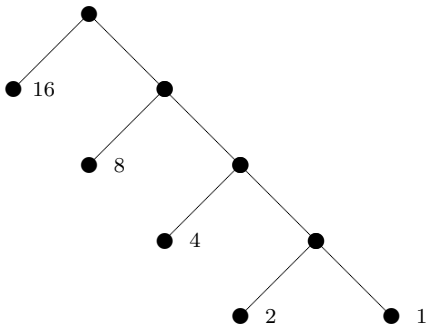
Из њега непосредно следи тврђење теореме.

7.5 Уобичајене грешке

Са редукцијама треба бити опрезан. Навешћемо примере уобичајених грешака које се могу направити при покушају редукције. Најчешћа грешка је да се редукција изведе у погрешном смеру, а до ње долази углавном при редукцијама за добијање доњих граница. Нека је потребно редукцијом доказати да је неки проблем P тежак бар колико други проблем Q, чију сложеност знамо. Тада треба поћи од *произвољног* улаза за проблем Q, и показати да се он може решити црном кутијом за решавање проблема P. Размотримо, на пример, следећи покушај редукције проблема сортирања на проблем компресије података Хафмановим кодом (оптималним префиксним кодом). Циљ је показати да је $\Omega(n \log n)$ доња граница сложености за Хафманово кодирање.

Полази се од запажања да, ако су учестаности знакова јако различите, онда кодно стабло постаје толико неуравнотежено да се може искористити за сортирање учестаности, видети пример на слици 7.3. У том случају су знакови у кодном стаблу поређани по опадајућим учестаностима (са најчешћим знаком најближим корену стабла). То управо значи да се Хафманово кодирање може искористити за сортирање учестаности. Према томе, формирање кодног стабла је тешко бар толико колико и сортирање, па је тиме изгледа доказана доња граница сложености $\Omega(n \log n)$.

Грешка у доказу крије се у чињеници да смо пошли од специјалног улаза за проблем сортирања: разматрали смо само такве учестаности, које су значајно раздвојене. Да би се дошло до доње границе сложености за проблем кодирања, мора се поћи од *произвољног улаза* за проблем сортирања:



Слика 7.3: Хафманово кодно стабло кад се учестаности знакова драстично разликују.

треба доказати да се произвољни бројеви могу сортирати помоћу алгоритма за Хафманово кодирање.

Испоставља се да се у овом случају грешка може исправити. Идеја је да се утроши још мало времена да би се променио (произвољни!) улаз за проблем сортирања, тако да се он може искористити као улаз за кодирање. Нека је улаз низ различитих природних бројева $X = x_1, x_2, \ldots, x_n$. Може се претпоставити да су бројеви различити, јер доња граница за сортирање важи и за инстанце које се састоје од различитих бројева (шта више, доња граница је доказана за различите бројеве). Кодно стабло Хафмановог кода за ове учестаности може бити произвољног облика, па је претходно разматрање неприменљиво. Међутим, сваки број x_i може се заменити учестаношћу $y_i = 2^{x_i}$. Пошто за сваки природни број m важи $2^m > \sum_{i<m} 2^i$, кодно стабло имаће облик као на слици 7.3 (свака учестаност у низу већа је од збира свих учестаности мањих од ње). Према томе, алгоритам за Хафманово кодирање може се искористити за сортирање бројева y_i. Потребно је још уверити се да број операција изведених при редукцији није недозвољено велики. Степеновање може да садржи велики број операција, али то овде није битно, јер доња граница за сортирање обухвата само упоређивања. Према томе, доказано је да, ако се претпостави модел стабла одлучивања, формирање Хафмановог кодног стабла у најгорем случају захтева $\Omega(n \log n)$ упоређивања (потенцијално је могуће брже формирати стабло алгоритмом који није обухваћен моделом стабла одлучивања).

Код алгоритама добијених редукцијом важно је да сама редукција не унесе значајну неефикасност. Размотримо проблем ранца и његово уопштење кад се копије сваког од предмета могу у ранац ставити неограничени број пута. Директна редукција уопштеног проблема на полазни (у коме се сваки предмет може искористити само једном) је следећа. Нека је величина ранца K. У ранац може да стане највише K/s_i копија предмета величине s_i. Према томе, у полазном проблему може се сваки предмет заменити са K/s_i предмета исте величине у полазном проблему. Иако је ова редукција коректна, она није ефикасна, јер је величина проблема значајно повећана.

7.6 Резиме

Увек је корисно разматрати сличности између проблема. Анализом разлика и сличности између два проблема, обично се стиче боља представа о оба проблема. Кад се наиђе на нови проблем, у већини случајева је корисно упитати се да ли је он сличан неком већ познатом проблему. Понекад сличност између два проблема постаје видљива тек по извршењу компликоване редукције. Посебно су интересантне редукције између матричних и графовских алгоритама. У овом поглављу видели смо више примера редукција.

Линеарно и целобројно програмирање су овде изложени врло кратко. То су веома важни проблеми, и корисно је да се са њима детаљније упозна свако кога интересују алгоритми.

Глава 8

Гранање са одсецањем

8.1 Минимизирање максималне суме узастопна три броја на кругу

Посматрајмо наредну комбинаторну загонетку која укључује дванаест бројева на сату: могу ли се бројеви разместити на кругу тако да ниједна тројка узастопних бројева нема суму већу од 21? Вредност 21 представља најмању вредност коју максимална сума узастопних тројки може да има. Један начин да се то установи је да се нађе максимална сума узастопна три броја за свих $11! = 39916800$ суштински различитих распореда ових 12 бројева на кругу (положај једног од њих се може фиксирати, што не утиче на резултат); од тих максималних сума треба изабрати најмању. У наставку ћемо видети како се простор претраге приликом решавања овог проблема у општем случају, са n бројева на кругу, може битно смањити применом гранања са одсецањем.

Програм може бити користан за разматрање технике гранање са одсецањем приказујући снагу, али и ограничења приступа чисте претраге [7]. Овај проблем може се решити генерисањем свих пермутација, а затим систематичном провером да ли поједине пермутације задовољавају тражене услове. Пермутације се могу генерисати рекурзивним алгоритмом и у алгоритам се може уградити механизам одсецања. Оваква имплементација би могла да детектује тренутак када се у парцијалној пермутацији појавила тројка узастопних елемената са сумом већом од задате, на основу чега би се извршило одсецање (јер ни једна од пермутација изведених из ове парцијалне пермутације не задовољава услов). Овим би се значајно убрзала претрага. Ипак, недостатак овог приступа би био тај што би се у стаблу одлучивања претрага спуштала дубоко низ стабло за грану која не може да генерише решење јер су бројеви на почетку пермутације премали (па не постоји пермутација која би садржала преостале велике бројеве и која би задовољавала услов проблема).

Посматрајмо сада нешто општији проблем одређивања пермутације бројева од 1 до n са најмањим максималним збиром три узастопна члана. Претпоставимо да су позицијама од 0 до k у низу x придружени бројеви – елементи пермутације, а позицијама од $k+1$ до $n-1$ још увек не (видети слику 8.1). Сума сваке узастопне тројке не сме да буде већа од $MaxSum$. Постоји $n-k+1$ група непознатих сума тројки.

$\cdots$	x_{k-1}	x_k	x_{k+1}	$\cdots$	x_{n-1}	x_0	x_1	$\cdots$
$\cdots$	познато	познато	непознато	$\cdots$	непознато	познато	познато	$\cdots$

Слика 8.1: Графички приказ кружне листе позиција и елемената пермутација који су им додељени: за позиције од 0 до k знамо, а за позиције од $k+1$ до $n-1$ не знамо додељене елементе пермутације.

Нека R означава суму бројева који још увек нису додељени ниједној позицији. Тада важи следеће: x_{k-1} се појављује у једној тројци, x_k у две тројке, свака од вредности x_i, $i = k+1, \ldots, n-1$, појављује се у три тројке, x_0 се појављује у две, а x_1 у једној тројци. Збир непознатих сума узастопних тројки је $x_{k-1} + 2x_k + 3R + 2x_0 + x_1$. Ни једна узастопна тројка не сме да има збир већи од $MaxSum$, па просечна вредност $(x_{k-1} + 2x_k + 3R + 2x_0 + x_1)/(n-k+1)$ непознатих сума узастопних тројки мора да буде мања или једнака од $MaxSum$. Дакле, могуће је добити решење са задатим почетком премутације ако важи услов

$$(n-k+1)MaxSum \geq x_{k-1} + 2x_k + 3R + 2x_0 + x_1.$$

Овакво ограничење је посебно ефикасно у проблемима у којима постоји мало валидних решења.

Глава 9

NP **комплетност**

9.1 Доказ непостојања приближног алгоритма за решавање (општег) проблема

О проблему трговачког путника (traveling salesman problem, скраћено, TSP) је било речи раније. Нека је $G = (V, E)$ комплетни неусмерени тежински граф, при чему је тежина (цена) гране $(u, v) \in E$ означена са $c(u, v) \geq 0$. Проблем је пронаћи Хамилтонов циклус са најмањом ценом.

У многим практичним проблемима најјефтинији начин да се посети чвор v из чвора u је управо кроз грану од u до v, уместо преко неких других грана који чине пут између та два чвора. Овај појам можемо формализовати тако што кажемо да функција цене c задовољава *неједнакост троугла* ако за свака три чвора $u, v, w \in V$ важи

$$c(u, w) \leq c(u, v) + c(v, w).$$

Специјално, овај услов је испуњен ако су чворови графа тачке у равни, а дужине грана су еуклидска растојања крајева грана (тзв. еуклидски проблем трговачког путника). Као што смо видели, постоје приближни алгоритми полиномијалне сложености за решавање еуклидског проблема трговачког путника, такви да проналазе Хамилтонов циклус дужине највише $\rho \cdot opt$, где је ρ једнако 2 или 3/2, а opt је минимална дужина Хамилтоновог циклуса. У општем случају за решавање TSP не постоји приближни алгоритам полиномијалне сложености за било које $\rho > 1$, осим ако није $P = NP$. Ово тврђење предмет је теореме 9.1.

Теорема 9.1 (Непостојање приближног алгоритма за решавање TSP)**.** *Ако је $P \neq NP$, онда за било коју константу $\rho \geq 1$ важи да не постоји приближни алгоритам полиномијалне временске сложености за решавање (општег) проблема трговачког путника у полиномијалној временској сложености са апроксимацијом до на фактор ρ (односно, да је циклус који приближни алгоритам враћа највише ρ пута дужи од дужине оптималног Хамилтоновог циклуса).*

Доказ. Доказ ћемо извести извођењем контрадикције. Претпоставимо супротно, односно, да за неки број $\rho \geq 1$, постоји приближни алгоритам A полиномијалне временске сложености са апроксимацијом до на фактор ρ. Без губитка на

општости, претпоставимо да је ρ целобројна вредност, уз заокруживање уколико је то неопходно. Сада ћемо показати како је могуће искористити алгоритам A за решавање инстанци проблема Хамилтоновог циклуса алгоритмом полиномијалне временске сложености. Како знамо да је проблем проналажења Хамилтоновог циклуса NP-комплетан проблем, следи да ако можемо да га решимо алгоритмом полиномијалне временске сложености, онда је $P = NP$.

Нека је $G = (V, E)$ задати граф – инстанца проблема проналажења Хамилтоновог циклуса; циљ је утврдити да ли G садржи Хамилтонов циклус користећи хипотетички приближни алгоритам A. Полазећи од графа G може се формирати инстанца проблема TSP на следећи начин. Нека је $g' = (V, E')$ комплетан граф са скупом чворова V, односно,

$$E' = \{(u, v) \mid u, v \in V \text{ и } u \neq v\}.$$

Придружујемо цену свакој грани из E' на следећи начин:

$$c(u, v) = \begin{cases} 1, & \text{ако } (u, v) \in E, \\ \rho|V| + 1, & \text{иначе.} \end{cases}$$

Јасно је да је сложеност формирања тежинског графа G' од полазног графа G полином од $|V|$ и $|E|$.

Размотримо проблем трговачког путника (G', c). Ако почетни граф G има Хамилтонов циклус H, онда функција цене c придружује свакој грани из H цену 1, одакле се добија да (G', c) садржи Хамилтонов циклус цене $|V|$. Са друге стране, ако G не садржи Хамилтонов циклус, онда било који Хамилтонов циклус у G' мора да садржи неку грану која се не налази у E. Међутим, произвољан такав циклус који садржи неку грану која се не налази у E има цену барем

$$\begin{aligned}(\rho|V| + 1) + (|V| - 1) &= \rho|V| + |V| \\ &= (\rho + 1)|V|.\end{aligned}$$

Дакле, цена Хамилтоновог циклуса у G' је барем за фактор $\rho + 1$ већа од цене Хамилтоновог циклуса у G' који јесте Хамилтонов циклус у G.

Применимо приближни алгоритам A полиномијалне сложености (за који смо претпоставили да постоји) на инстанцу (G', c) проблема TSP. С обзиром да алгоритам A даје као резултат Хамилтонов циклус цене не веће од ρ пута цене оптималног пута, ако G садржи Хамилтонов циклус, онда A мора да га врати. Ако G нема Хамилтонов циклус, онда A проналази у G' Хамилтонов циклус дужине бар $(\rho + 1)V > \rho V$. Дакле, алгоритам A полиномијалне сложености може се искористити за утврђивање да ли у G постоји Хамилтонов циклус. Из овог закључка следи да је $P = NP$, супротно претпоставци теореме.

9.2 Доказ NP-комплетности проблема "збир подскупа"

Сада ћемо размотрити један аритметички NP-комплетан проблем под називом *збир подскупа*. Претпоставља се да је дат коначан скуп S позитивних целих бројева и цели број $t > 0$. Треба пронаћи да ли постоји подскуп

$S' \subseteq S$ такав да је збир његових елемената једнак броју t. На пример, ако је $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ и $t = 138457$, онда подскуп $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ скупа S задовољава тражени услов.

Као и у вези са било којим аритметичким проблемом, требало би се подсетити да наше стандардно кодирање претпоставља да су улазни цели бројеви кодирани као бинарни. Са овом претпоставком на уму, можемо показати да је проблем збира подскупа NP-комплетан.

Теорема 9.2 (NP-комплетност проблема збир подскупа)**.** *Проблем збир подскупа је* NP*-комплетан.*

Доказ. Прво, јасно је да се у полиномијалном времену може проверити да ли је збир елемената подскупа S' скупа S једнак вредности t. Сада ћемо показати да је проблем $3SAT$ полиномијално сводљив на проблем збира подскупа. Нека је дата Булов израз у коњуктивној нормалној форми F над променљивим $x_1, x_2, \ldots, x_n$ са клаузама $C_1, C_2, \ldots, C_k$, где свака клауза садржи тачно три различита литерала. Редукциони алгоритам конструише инстанцу (S, t) проблема збира подскупа такву да је израз F задовољив ако и само ако постоји подскуп скупа S чија је сума елемената једнака t. Без смањења општости, претпоставићемо следеће две особине израза F. Прво, ниједна клауза не садржи променљиву и њену негацију, јер је таква клауза аутоматски задовољена у било којој валуацији променљивих. Друго, свака променљива се појављује у барем једној клаузи, јер није важно која вредност је додељена променљивим које се не налазе у клаузама.

Редукција додаје два броја у скуп S за сваку променљиву x_i и два броја у скуп S за сваку клаузу C_j. Бројеви се конструишу у основи 10, где сваки број има $n + k$ цифара и свака цифра одговара или једној променљивој или једној клаузи. Основа 10 има својство које је неопходно за спречавање преноса са ниже позиције на вишу.

Конструишемо скуп S и циљну вредност t на следећи начин (види слику 9.1). Означићемо сваку позицију цифре или променљивом или клаузом. Тачно k најмање значајних цифара су означене клаузама и тачно n најзначајних цифара су означене променљивима.

- Циљ t има број 1 у свакој цифри означеној променљивом и има број 4 у свакој цифри означеној клаузом.

- За сваку променљиву x_i, скуп S садржи два цела броја v_i и v'_i. Сваки од бројева v_i и v'_i има 1 као цифру означену са x_i и број 0 у цифрама означених осталим променљивама. Ако се литерал x_i појављује у клаузи C_j, онда цифра означена клаузом C_j у v_i садржи број 1. Ако се литерал $\neg x_i$ појављује у клаузи C_j, онда цифра означена клаузом C_j у v'_i садржи број 1. Све остале цифре означене клаузама у v_i и v'_i имају број 0.

  Тврдимо да су све вредности v_i и v'_i у скупу S јединствене. За $l \neq i$, ниједна од вредности v_l и v'_l не може бити једнака ни једној од вредности v_i и v'_i, што се види ако се посматра њихових највиших n цифара. Даље, из претпоставки следи да је $v_i \neq v'_i$. Ова два броја се не разликују ако се посматра само горњих n цифара, али се ова два броја увек разликују бар по једној од доњих k најнижих цифара. Ако

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Слика 9.1: Пример редукције проблема $3SAT$ на проблем збир подскупа. Формула F је $C_1 \wedge C_2 \wedge C_3 \wedge C_4$, где је $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$ и $C_4 = (x_1 \vee x_2 \vee x_3)$. За $x_1 = 0, x_2 = 0, x_3 = 1$ вредност израза је 1, па је израз F задовољив. Бројеви који улазе у подскуп S' су означени тамно сивом бојом.

би вредност v_i и v'_i биле једнаке, онда би променљиве x_i и $\neg x_i$ морале да се појаве у истом скупу клауза. Међутим, ми претпостављамо да ниједна клауза не садржи обе променљиве x_i и $\neg x_i$, али и да се или x_i или $\neg x_i$ појављују у некој клаузи, тако да мора да постоји клауза C_j за коју се одговарајуће цифре бројева v_i и v'_i разликују.

- За сваку клаузу C_j, скуп S садржи два цела броја s_j и s'_j. Свака од вредности s_j и s'_j има све цифре 0, сем цифре која одговара клаузи C_j. Исто важи и за бројеве $0, s_j, s'_j, s_j + s'_j$, чија цифра која одговара клаузи C_j је редом $0, 1, 2, 3$.

  Слично као у претходном случају, може се показати да су све вредности s_j и s'_j јединствене у S.

Приметимо да сума цифара на било којој позицији може да буде највише 6, што се може десити на позицијама означеним клаузама (три јединице које одговарају вредности v_i и v'_i, и додатна јединица и двојка које одговарају вредностима s_j и s'_j, редом). Одавде следи да, интерпретирањем ових цифара у основи 10, није могуће да се појави пренос са неке позиције на вишу позицију.

Сложеност описане редукције је полиномијална. Скуп S садржи $2n+2k$ вредности, од којих свака има $n+k$ цифара; свака цифра се израчунава за време $O(n+k)$. Циљ t има $n+k$ цифара и редукција формира сваку од ових цифара за време $O(1)$.

Сада ћемо показати да је формула F задовољива ако и само ако постоји подскуп S' скупа S чија сума износи t. Прво, претпоставимо да је формула

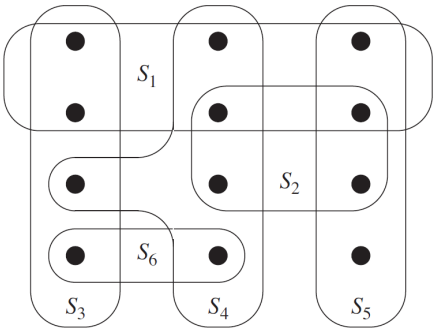
F задовољива у некој валуацији v. За свако $i = 1, 2, \ldots, n$ ако је $v_F(x_i) = 1$, онда треба укључити v_i у S'. У противном, укључити v'_i. Другим речима, у скуп S' укључујемо тачно вредности v_i и v'_i које одговарају литералима са вредношћу 1 у задовољавајућој валуацији. Укључивањем или v_i или v'_i, али не и обе, за свако i, и стављајући цифру 0 на све позиције означене променљивима у свим s_j и s'_j, видимо да за сваку позицију означену променљивом, збир вредности у S' мора бити 1, што се поклапа са одговарајућим цифрама у t. С обзиром да је произвољна клауза задовољена, она садржи бар један литерал са вредношћу 1. Дакле, свака цифра означена клаузом има барем једну јединицу која доприноси суми у оквиру v_i или v'_i у S'. Прецизније, један, два или три литерала могу бити 1 у свакој клаузи, па тако свака позиција означена клаузом има суму 1, 2 или 3 од вредности v_i и v'_i у S'. Циљ 4 се добија на свакој позицији означеној клаузом C_j тако што се у скуп S' укључује одговарајући непразан подскуп слек променљивих $\{s_j, s'_j\}$. Како смо поклопили циљ по свим цифрама збира, и не постоји пренос, вредности скупа S' се сумирају на t.

Сада претпоставимо да постоји подскуп S' скупа S са сумом једнаком t. Подскуп S' мора да укључи тачно једну од вредности v_i и v'_i за свако $i = 1, 2, \ldots, n$; у противном збир цифара бројева из S' не би био једнак 1. Ако $v_i \in S'$, онда постављамо $v_F(x_i) = 1$. Иначе је $v'_i \in S'$, односно, $v_F(x_i) = 0$. Тврдимо да за сваку клаузу C_j, за $j = 1, 2, \ldots, k$, важи да је задовољена оваквом валуацијом. Да бисмо ово показали, приметимо да, с обзиром да је сума цифара на позицији означеној клаузом C_j једнака 4, подскуп S' мора да садржи бар један од бројева v_i или v'_i, који имају цифре 1 на позицији означеној клаузом C_j, јер се заједнички допринос слек променљивих s_j и s'_j тој цифри збира највише 3. Ако скуп S' садржи v_i која има цифру 1 на позицији означеној C_j, онда се литерал x_i појављује у клаузи C_j. Ако скуп S' садржи v'_i која има цифру 1 на позицији означеној C_j, онда се литерал $\neg x_i$ појављује у клаузи C_j. С обзиром да смо поставили $v_F(x_i) = 0$ када је $v'_i \in S'$, клауза C_j је свакако тачна. Дакле, све клаузе израза F су тачне, што значи да је израз F задовољив.

9.3 Приближни алгоритми за решавање проблема минималног покривања скуп

Проблем минималног покривања скупа (set covering problem, скраћено, SCP) представља оптимизациони проблем који моделује многе проблеме у вези са алокацијом ресурса. Његов одговарајући проблем одлучивања генерализује NP-комплетан проблем покривања скупа чворова графа, и сам је такође NP-комплетан. Приближни алгоритми за решавање проблема покривања скупа чворова графа се не могу применити на овај општи случај, па је неопходно покушати другим апроксимацијама. У наставку ћемо анализирати једноставну похлепну хеуристику са логаритамским фактором апроксимације. Другим речима, како се величина инстанце повећава, величина приближног решења може да расте, релативно у односу на оптимално решење. Ипак, с обзиром да логаритамска функција споро расте, приближни алгоритам може да дâ корисне резултате.

Инстанца $(X, \mathcal{F})$ проблема *минималног покривања скупа* састоји се од коначног скупа X и фамилије $\mathcal{F}$ подскупова скупа X, тако да сваки елемент



Слика 9.2: Инстанца $(X, \mathcal{F})$ проблема минималног покривача скупа, где се скуп S састоји од 12 елемената означених црним тачкама и $\mathcal{F} = \{S_1, S_2, \ldots, S_6\}$. Минимални покривач скупа X је $\mathcal{C} = \{S_3, S_4, S_5\}$, кардиналности 3. Описани похлепни алгоритам проналази покривач кардиналности 4 који се састоји од скупова S_1, S_4, S_5 и S_3 или скупова S_1, S_4, S_5 и S_6, у том редоследу.

скупа X припада барем једном подскупу из $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S.$$

Кажемо да подскуп $C \subseteq X$ *покрива* своје елементе. Проблем је пронаћи подскуп $\mathcal{C} \subseteq \mathcal{F}$ најмање кардиналности чији елементи покривају цео скуп X:

$$X = \bigcup_{S \in \mathcal{C}} S. \tag{9.1}$$

Ако нека фамилија $\mathcal{C}$ задовољава једнакост 9.1, онда кажемо да она *покрива* скуп X. На слици 9.2 приказан је пример проблема минималног покривача скупа. На слици 9.2 минимални покривач скупа је кардиналности 3.

Похлепни приступ решавању овог проблема може се описати на следећи начин: у свакој фази треба одабрати скуп S који покрива највећи број до сада непокривених елемената. Овај алгоритам може се описати следећим кодом.

У примеру на слици 9.2, похлепни алгоритам додаје редом у $\mathcal{C}$ скупове S_1, S_4, S_5, а затим бира један од скупова S_3 или S_6.

Сложеност. Могуће је једноставно имплементирати алгоритам тако да му сложеност буде полином од $|X|$ и $|\mathcal{F}|$. С обзиром на то да је број итерација петље ограничен вредношћу $\min\{|X|, |\mathcal{F}|\}$ и је сложеност тела петље $O(|X||\mathcal{F}|)$, укупна сложеност једноставне имплементације је $O(|X||\mathcal{F}| \min\{|X|, |\mathcal{F}|\})$. Напоменимо да се алгоритам може имплементирати тако да му сложеност буде $O(\sum_{S \in \mathcal{F}} |S|)$.

Анализа. Сада ћемо показати да похлепни алгоритам враћа покривач скупа који није много већи од минималног покривача скупа. Нека је $H(d)$ d-та парцијална сума хармонијског реда, $H(d) = \sum_{i=1}^{d} 1/i$ и нека је $H(0) = 0$

Алгоритам Pohlepni_pokrivac_skupa$(X, \mathcal{F})$;
Улаз: скуп X чији минимални покривач тражимо и фамилија $\mathcal{F}$ подскупова скупа X.
Излаз: C (приближни минимални покривач скупа X).
begin
    $U := X$
    $\mathcal{C} := \emptyset$
    while U је непразан **do**
        Одабери скуп $S \in \mathcal{F}$ који максимизује вредност $|S \cap U|$
        $U := U \setminus S$
        $\mathcal{C} := \mathcal{C} \cup \{S\}$
end

Слика 9.3: Похлепни алгоритам за покривање скупа.

Теорема 9.3 (Апроксимативна моћ похлепног алгоритма за решавање SCP)**.** *Приказани похлепни алгоритам је приближни алгоритам полиномијалне временске сложености са фактором апроксимације*

$$\rho(n) = H\left(\max_{S \in \mathcal{F}}\{|S|\}\right).$$

Доказ. Већ смо показали да је приказани алгоритам полиномијалне временске сложености. Да бисмо показали да је алгоритам приближни алгоритам са фактором апроксимације $\rho(n)$, додељујемо цену 1 сваком скупу који је одабран од стране алгоритма и равномерно расподељујемо ову цену свим елементима који се овим скупом покривају први пут. Ове цене биће искоришћене за извођење жељеног односа између кардиналности минималног скупа покривача $\mathcal{C}^*$ и кардиналности покривача $\mathcal{C}$ добијеног као излаз алгоритма. Прецизније, нека S_i означава i-ти изабрани скуп од стране алгоритма; алгоритам задужује цену 1 када додаје скуп S_i у $\mathcal{C}$. Ову цену прерасподељујемо по елементима који се покривају по први пут скупом S_i. Означимо са c_x цену расподељену елементу $x \in X$. Сваком елементу се цена додељује тачно једном, онда када се он покрива по први пут. Ако је x покривен по први пут скупом S_i, онда је

$$c_x = \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}.$$

Сваки корак у алгоритму додељује цену 1, те је стога укупна цена $|\mathcal{C}|$ фамилије $\mathcal{C}$ једнака

$$|\mathcal{C}| = \sum_{x \in X} c_x. \tag{9.2}$$

Сваки елемент $x \in X$ је у барем једном скупу минималног покривача $\mathcal{C}^*$, те је стога

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x. \tag{9.3}$$

Комбиновањем једнакости 9.2 и неједнакости 9.3 добијамо неједнакост

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x. \tag{9.4}$$

Остатак доказа теореме заснива се на наредној кључној неједнакости, коју ћемо доказати у наставку. За произвољан скуп $S \in \mathcal{F}$ важи неједнакост

$$\sum_{x \in S} c_x \leq H(|S|). \tag{9.5}$$

Комбиновањем неједнакости 9.4 и 9.5 добија се

$$\begin{aligned} |\mathcal{C}| &\leq \sum_{S \in \mathcal{C}^*} H(|S|) \\ &\leq |\mathcal{C}^*| \cdot H\left(\max_{S \in \mathcal{F}}\{|S|\}\right), \end{aligned}$$

што је и требало доказати.

Остало је да докажемо неједнакост 9.5. Нека је $S \in \mathcal{F}$ произвољан скуп и нека за произвољно $i = 1, 2, \ldots, |\mathcal{C}|$

$$u_i = |S \setminus (S_1 \cup S_2 \cup \ldots \cup S_i)|$$

означава број елемената скупа S који су остали непокривени након што је алгоритам одабрао скупове $S_1, S_2, \ldots, S_i$. Дефинишемо $u_0 = |S|$ као број елемената скупа S, с обзиром да су на почетку алгоритма сви елементи непокривени. Нека је k најмањи индекс i такав да је $u_i = 0$, односно такав индекс да је сваки елемент скупа S покривен барем једним од скупова $S_1, S_2, \ldots, S_i$ и да је барем неки елемент скупа S непокривен унијом $S_1 \cup S_2 \cup \ldots \cup S_{i-1}$. Јасно је да је $u_{i-1} - u_i \geq 0$ елемената скупа S по први пут покривено скупом S_i, због чега је $u_{i-1} \geq u_i$, за свако $i = 1, 2, \ldots, k$. Дакле,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}.$$

Приметимо да је

$$\begin{aligned} |S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| &\geq |S \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| \\ &= u_{i-1}, \end{aligned}$$

с обзиром да похлепан избор скупа S_i гарантује да S не може да покрије више нових елемената него што то чини скуп S_i (у супротном, алгоритам би изабрао S уместо S_i). Као последицу добијамо

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

Горња граница за ову суму може се добити на следећи начин:

$$\begin{aligned}
\sum_{x\in S} c_x &\le \sum_{i=1}^{k}(u_{i-1}-u_i)\cdot\frac{1}{u_{i-1}} = \sum_{i=1}^{k}\sum_{j=u_i+1}^{u_{i-1}}\frac{1}{u_{i-1}} \\
&\le \sum_{i=1}^{k}\sum_{j=u_i+1}^{u_{i-1}}\frac{1}{j} && \text{(јер је } j \le u_{i-1}\text{)} \\
&= \sum_{i=1}^{k}\left(\sum_{j=1}^{u_{i-1}}\frac{1}{j}-\sum_{j=1}^{u_i}\frac{1}{j}\right) = \sum_{i=1}^{k}(H(u_{i-1})-H(u_i)) \\
&= H(u_0)-H(u_k) && \text{(јер је овај ред “телескопски”)} \\
&= H(u_0)-H(0) = H(u_0) = H(|S|),
\end{aligned}$$

чиме је завршен доказ неједнакости 9.5.

Чињеница да је фактор апроксимације алгоритма *Pohlepni_pokrivac_skupa* логаритамска функција од броја елемената скупа X следи из теореме 9.3 и раније изведене неједнакости

$$H(n) = \sum_{k=1}^{n}\frac{1}{k} \le \ln n + 1. \tag{9.6}$$

Теорема 9.4 (Логаритамска апроксимативна моћ алгоритма за приближно решавање SCP)**.** *Фактор апроксимације алгоритма Pohlepni_pokrivac_skupa износи* $O(\log|X|)$.

Глава 10

Паралелни алгоритми

10.1 Увод

Паралелно израчунавање више није егзотична област, и већ дуже време је на главном правцу развоја рачунарства. Развија се врло брзо, чак и у односу на друге рачунарске области. У употреби је више типова паралелних рачунара, са бројем процесора у опсегу од 2 до 65536, и већим. Разлике између различитих постојећих рачунара, чак и са аспекта необавештеног корисника, врло су велике. Немогуће је усвојити један општи модел израчунавања који би обухватао све паралелне рачунаре

У овом поглављу нису покривене све (па чак ни већина) области повезаних са паралелним израчунавањем. Приказани су примери коришћења појединих модела израчунавања и различите технике. Циљ је стицање представе о паралелним алгоритмима и упознавање са потешкоћама везаним за њихову конструкцију. Почиње се са заједничким карактеристикама паралелних алгоритама. Затим се укратко описују неки основни модели паралелног израчунавања, а на крају се наводе примери алгоритама и техника.

Основне мере сложености за секвенцијалне алгоритме су време извршавања и величина коришћене меморије. Ове мере су важне и код паралелних алгоритама, али се мора водити рачуна и о другим ресурсима, посебно о броју процесора. Постоје проблеми који су суштински секвенцијални, који се не могу "паралелизовати" чак ни ако је на располагању неограничени број процесора. Ипак, већина осталих проблема може се до неког степена паралелизовати. Што више процесора се користи — до неке границе — алгоритам се брже извршава. Важно је проучавати ограничења паралелних алгоритама, и бити у стању окарактерисати проблеме за које постоје врло брза паралелна решења. Пошто је број процесора ограничен, исто тако је важно да се процесори ефикасно користе. Следећи важан елеменат је комуникација између процесора. Често је више времена потребно да два процесора размене податке, него да се изврше једноставне операције са подацима. Поред тога, трајање размене података може да зависи од "удаљености" два процесора у рачунару. Према томе, важно је минимизирати комуникацију и организовати је на ефикасни начин. Следеће важно питање је синхронизација, која је велики проблем код паралелних алгоритама кад се извршавају на независним машинама, повезаним неком мрежом за комуникацију. Такви

алгоритми се обично зову **дистрибуирани алгоритми**. Њих овде нећемо разматрати; ограничићемо се на моделе са потпуном синхронизацијом.

Неки модели паралелног израчунавања садрже ограничење да сви процесори у једном кораку извршавају једну исту инструкцију (над евентуално различитим подацима). Паралелни рачунари са оваквим ограничењем зову се SIMD (скраћеница од Single–Instruction Multiple–Data) рачунари. Паралелни рачунари код којих сваки процесор може да извршава различити програм зову се MIMD (скраћеница од Multiple–Instruction Multiple–Data) рачунари. Уколико се не нагласи другачије, претпоставља се да су рачунари о којима је реч MIMD рачунари.

10.2 Модели паралелног израчунавања

Детаљан преглед модела паралелних рачунара захтевао би више простора. Спомењућемо само основне моделе, са нагласком на оне који се користе у овом поглављу. У овом одељку изложићемо нека општа разматрања и дефиниције који се односе на многе моделе. Сваки од следећих одељака покрива један од типова модела, садржи његов детаљнији опис и примере алгоритама.

Време извршавања алгоритма означаваћемо са $T(n,p)$, где је n величина улаза, а p број процесора. Однос

$$S(p) = T(n,1)/T(n,p)$$

зове се **убрзање** алгоритма. Паралелни алгоритам је најефикаснији кад је $S(p) = p$, тј. кад алгоритам достиже **савршено убрзање**. За вредност $T(n,1)$ треба узети *најбољи* познати секвенцијални алгоритам. Важна мера искоришћености процесора је **ефикасност** паралелног алгоритма, која се дефинише изразом

$$E(n,p) = \frac{S(p)}{p} = \frac{T(n,1)}{pT(n,p)}.$$

Ефикасност је однос времена извршавања на једном процесору (кад извршава секвенцијални алгоритам) и *укупног времена* извршавања на p процесора (укупно време је стварно протекло време помножено бројем процесора). Ефикасност указује на удео процесорског времена, које се ефективно користи у односу на секвенцијални алгоритам. Ако је $E(n,p) = 1$, онда је количина рачунања обављеног на свим процесорима у току извршавања алгоритма једнака количини рачунања коју захтева секвенцијални алгоритам. У том случају постигнуто је оптимално искоришћење процесора. Постизање оптималне ефикасности је ретко, јер се у паралелним алгоритмима морају извршити нека допунска израчунавања, која нису потребна код секвенцијалног алгоритма. Један од основних циљева је максимизирање ефикасности.

При конструкцији паралелног алгоритма могло би се фиксирати p, у складу са бројем процесора на располагању, и покушати са минимизирањем $T(n,p)$. Али недостатак оваквог приступа је у томе што би он могао да захтева нови алгоритам, кад год се промени број процесора. Згодније би било конструисати алгоритам који ради за што је могуће више различитих вредности p. Размотрићемо сада како трансформисати алгоритам који ради за неку вредност p, у алгоритам за мању вредност p, без значајне промене ефикасности. У општем случају, алгоритам са $T(n,p) = X$ може се трансформисати

у алгоритам са $T(n, p/k) \simeq kX$, за произвољну константу $k > 1$. Другим речима, може се користити за фактор k мање процесора, чије је време рада онда дуже за фактор k. Модификовани алгоритам може се конструисати заменом сваког корака полазног алгоритма са k корака, у којима један процесор *емулира* (паралелно) извршавање једног корака на k процесора. Овај принцип није увек применљив. На пример, могуће је да p није дељиво са k, или да алгоритам зависи од начина повезивања процесора (о чему ће бити речи у одељку 10.4), или да доношење одлуке о томе које процесоре емулирати захтева такође утрошак неког времена. Ипак, овај принцип, такозвани **принцип имитирања паралелизма**, врло је општи и користан. Он показује да се може смањити број процесора, не мењајући битно ефикасност. Ако полазни алгоритам (који је конструисан за велике p) има велико убрзање, онда се могу добити алгоритми који постижу приближно исто убрзање за било коју мању вредност p. Према томе, треба конструисати алгоритам са што бољим убрзањем за максимални број процесора, при чему ефикасност треба да буде добра (тј. блиска јединици). Затим, ако је на располагању мањи број процесора, и даље се може користити исти алгоритам. С друге стране, паралелни алгоритми са малом ефикасношћу су корисни само ако је на располагању велики број процесора. Претпоставимо, на пример, да имамо алгоритам са $T(n, 1) = n$ и $T(n, n) = \log_2 n$, односно са убрзањем $S(n) = n/\log_2 n$ и ефикасношћу $E(n) = 1/\log_2 n$. Претпоставимо да нам је на располагању $p = 256$ процесора и да је $n = 1024$. Време извршавања паралелног алгоритма је $T(1024, 256) = 4 \log_2 1024 = 40$ (уз претпоставку да је могуће имитирање паралелизма биће $T(n, p) = (n \log_2 n)/p)$, што је убрзање за фактор око 25 у односу на секвенцијални алгоритам. С друге стране, за $p = 16$ време извршавања је 640, што даје недовољно убрзање (мање од 2 са 16 процесора).

Модели паралелног израчунавања разликују се међусобно углавном по начину комуникације и синхронизације процесора. Разматраћемо само моделе који подразумевају потпуну синхронизацију и различите начине повезивања. Модели са **заједничком меморијом** претпостављају да постоји заједничка меморија са равномерним приступом, тако да сваки процесор може да приступи свакој променљивој за јединично време. Ова претпоставка о времену приступа независном од броја процесора и величине меморије није баш реална, али је добра апроксимација. Модели са заједничком меморијом разликују се по начину на који обрађују конфликте приликом приступа меморији. Поједине алтернативе размотрићемо у одељку 10.3.

Заједничка меморија је обично најједноставнији начин за моделирање комуникације, али начин који је најтеже хардверски реализовати. Други модели претпостављају да су процесори међусобно повезани посредством **мреже**. Мрежа рачунара се може представити графом, при чему чворови одговарају процесорима, а два чвора су повезана ако између одговарајућих процесора постоји директна веза. Сваки процесор обично има локалну меморију, којој може да приступа брзо. Комуникација се остварује порукама, које морају да прођу више директних веза да би дошле до одредишта. Према томе, брзина комуникације зависи од растојања између процесора који размењују поруке. Неколико различитих графова је анализирано у улози скелета мреже рачунара. Више популарних конфигурација наведено је у одељку 10.4.

Следећи модел који ћемо размотрити је модел **систоличког рачунања**.

Систоличка архитектура подсећа на покретну траку у фабрици. Подаци се крећу кроз процесоре равномерно, и том приликом се над њима изводе једноставне операције. Уместо да приступају заједничкој (или локалној) меморији, процесори добијају улазне податке од својих суседа, обрађују их, и прослеђују даље. Неки систолички алгоритми наведени су у одељку 10.5.

Коло је основни теоријски модел, који ће бити коришћен само за потребе илустрације. Коло је усмерени ациклички граф, у коме чворови одговарају једноставним операцијама, а гране показују кретање операнада. На пример, Булово коло је оно у коме су улазни степени чворова највише два, а све операције су Булове операције (дисјункција, конјункција или негација). Посебно су издвојени улазни (са улазним степеном нула) и излазни чворови (са излазним степеном нула). Дубина кола је дужина најдужег пута од неког улазног до неког излазног чвора. Дубина одговара времену извршавања паралелног алгоритма.

10.3 Алгоритми за рачунаре са заједничком меморијом

Рачунар са заједничком меморијом састоји се од више **процесора** и **заједничке меморије**. У овом одељку бавићемо се само потпуно синхронизованим алгоритмима. Претпостављамо да се израчунавање састоји од *корака*. У сваком кораку сваки процесор извршава неку операцију над подацима којима располаже, чита из заједничке меморије или пише у заједничку меморију (у пракси сваки процесор може да има и локалну меморију). Модели са заједничком меморијом разликују се по томе како обрађују меморијске конфликте. Модел EREW (Exclusive–Read Exclusive–Write) не дозвољава да два процесора истовремено приступају истој меморијској локацији. Модел CREW (Concurent–Read Exclusive–Write) дозвољава да више процесора истовремено читају са исте меморијске локације, али не дозвољава да два процесора истовремено пишу на исту локацију. На крају, модел (Concurent–Read Concurent–Write) не намеће никаква ограничења на приступ процесора меморији.

Модели EREW и CREW су добро дефинисани, али није јасно шта је резултат истовременог писања од стране два процесора на једну исту меморијску локацију. Има више начина за обраду истовремених писања. Најслабији CRCW модел, једини који ће бити овде разматран, дозвољава да више процесора истовремено пишу на исту локацију само ако записују исту вредност. Ако два процесора покушају да упишу истовремено различите вредности на исту локацију, прекида се са извршавањем алгоритма. Иако је то можда неочекивано, видећемо у одељку 10.3.2 да је овакав модел врло моћан. Друга могућност је претпоставити да су процесори нумерисани, и да, ако више процесора покушају истовремени упис на исту локацију, реализује се упис процесора са највећим редним бројем.

10.3.1 Паралелно сабирање

Започињемо са једноставним примером паралелног алгоритма за решавање проблема, који је на први поглед суштински секвенцијалан.

Проблем. Израчунати суму два n-битна бинарна броја.

Обичан секвенцијални алгоритам најпре сабира два бита најмање тежине, а онда у сваком кораку сабира по два бита, и збиру евентуално додаје пренос са ниже позиције. На први поглед немогуће је предвидети исход i-тог корака, све док се не саберу $i-1$ бита најмање тежине, јер пренос може, а не мора да постоји. Ипак, могуће је конструисати паралелни алгоритам.

Користимо индукцију по n. Прелаз са $n-1$ на n не може да буде од велике користи, јер води итеративном секвенцијалном алгоритму. Приступ заснован на разлагању често даје добре резултате код паралелних алгоритама, па и у овом случају, јер може да реши све мање потпроблеме паралелно. Претпоставимо да смо поделили проблем на два потпроблема величине $n/2$, тј. на сабирање левих и десних сабирака од по $n/2$ бита (због једноставности претпостављамо да је n степен двојке). Суме два пара бројева могу се израчунати паралелно. Ипак, још увек остаје проблем преноса. Ако сума делова мање тежине има пренос, мора се променити сума делова веће тежине.

Проблем решава запажање да постоје само две могућности — пренос постоји или не. Захваљујући томе, може се *појачати индуктивна хипотеза*, тако да обухвати оба случаја. Модификовани проблем гласи: пронаћи суму два броја, са и без преноса на најнижу позицију. Претпоставимо да смо решили модификовани проблем за оба пара сабирака, и тако добили четири броја N, N_P, V и V_P, који представљају суму нижег пара без преноса, исту суму са преносом, и одговарајуће суме за виши пар бројева, редом. За сваку од тих сума такође знамо да ли је при њеном израчунавању дошло до преноса. Укупну суму S (без преноса на најнижи бит) чине N са V или V_P, зависно од тога да ли је сума N произвела пренос. Укупна сума S_P са преносом добија се на исти начин, само се N замењује са N_P. Пренос укупне суме је пренос V_P.

Проблем величине n своди се на решавање два потпроблема величине $n/2$ и на извршавање константног броја корака за обједињавање два резултата. Пошто се оба потпроблема могу решити паралелно — претпоставка је да процесори могу да приступе различитим битовима независно — добија се диференцна једначина $T(n,n) = T(n/2,n/2) + O(1)$, чије је решење $T(n,n) = O(\log n)$. Поред тога, пошто су два потпроблема потпуно независна, овај алгоритам подразумева само модел EREW. Овај алгоритам није најбољи за паралелно сабирање али је добар пример једноставне паралелизације алгоритма. Кад за неки проблем постане јасно да се може ефикасно решити паралелно, решење се може даље побољшавати.

10.3.2 Алгоритми за налажење максимума

Проблем. Пронаћи највећи од n различитих бројева, задатих у низу у заједничкој меморији.

Овај проблем решићемо за два различита модела са заједничком меморијом, EREW и CRCW. Алгоритми за оба модела користе технике које се користе при решавању многих других проблема.

Модел EREW

Директни секвенцијални алгоритам за налажење максимума захтева $n-1$ упоређивање. Упоређивање се може схватити као партија коју играју два броја, у којој побеђује већи. Проблем налажења максимума је тада еквивалентан организовању турнира, у коме је победник највећи број у целом скупу. Ефикасан начин да се турнир организује паралелно је да се искористи стабло. Играчи се деле у парове за прво коло (при чему евентуално један играч не учествује, ако је укупан број играча непаран), победници се поново деле у парове, и тако даље до финала. Број кола је $\lceil \log_2 n \rceil$. Турнир се може трансформисати у паралелни алгоритам тако што се свакој партији додели процесор (процесор игра улогу судије у партији). Међутим, треба одбезбедити да сваки процесор зна бројеве – такмичаре. То се може постићи копирањем победника у партији на позицију са већим индексом од позиција два учесника партије. Прецизније, ако партију играју x_i и x_j, $j > i$, онда се већи од бројева x_i, x_j копира на позицију j. У првом колу процесор P_i упоређује x_{2i-1} са x_{2i} $(1 \leq i \leq n/2)$, и замењује их ако је потребно, тако да на већу позицију оде већи број. У другом колу процесор P_i упоређује x_{4i-2} са x_{4i} $(1 \leq i \leq n/4)$, и тако даље. Ако је на пример $n = 2^k$, онда у последњем, k-том колу, P_1 упоређује $x_{n/2}$ са x_n, и евентуално их замењује. Највећи број налазиће се на позицији n. Пошто сваки број у једном тренутку учествује само у једној партији, довољан је модел EREW. Време извршавања овог једноставног алгоритма је очигледно $O(\log n)$. Покушаћемо сада да смањимо број коришћених процесора.

Алгоритам који смо управо размотрили захтева $\lfloor n/2 \rfloor$ процесора, а број корака је $T(n, \lfloor n/2 \rfloor) = \lceil \log_2 n \rceil$. Пошто је за секвенцијални алгоритам $T(n,1) = n-1$, ефикасност овог паралелног алгоритма је $E(n, n/2) \simeq 2/\log_2 n$. Ако нам је ионако на располагању $\lfloor n/2 \rfloor$ процесора (на пример, ако је алгоритам за налажење максимума део другог алгоритма, коме је неопходно толико процесора), онда је овај алгоритам једноставан и ефикасан. Међутим, уз мали напор може се доћи до алгоритма са временом извршавања $O(\log n)$ и ефикасношћу $O(1)$.

Укупан број упоређивања потребних за овај алгоритам је $n-1$, исти као и за секвенцијални алгоритам. Разлог мале ефикасности лежи у томе што се већина процесора не користи у каснијим колима. Ефикасност се може побољшати смањивањем броја процесора и уравнотежавањем њиховог оптерећења на следећи начин. Претпоставимо да користимо само око $n/\log_2 n$ процесора. Улаз се може поделити у $n/\log_2 n$ група (са приближно $\log_2 n$ елемената у свакој групи) и затим свакој групи доделити по један процесор. У првој фази сваки процесор проналази максимум у својој групи користећи секвенцијални алгоритам, који се састоји од око $\log_2 n$ корака. После тога остаје да се одреди максимум отприлике $n/\log_2 n$ максимума, при чему сад има довољно процесора да се искористи турнирски алгоритам. Време извршавања турнира је $T(n, \lceil n/\log_2 n \rceil) \simeq 2\log_2 n$. Одговарајућа ефикасност је $E(n) \simeq 1/2$. Покушаћемо сада да формализујемо ову идеју, која омогућује уштеду на броју процесора.

За алгоритам кажемо да је **статички** ако се унапред зна придруживање процесора операцијама. Дакле, унапред знамо за сваки корак i алгоритма и за сваки процесор P_j операцију и аргументе које P_j користи у кораку i. Алгоритам за налажење максимума је пример статичког алгоритма, јер се

унапред знају индекси учесника у свакој партији.

Лема 10.1 (Брентова лема)**.** Ако постоји статички EREW алгоритам са $T(n,p) = O(t(n))$, такав да је укупан број корака (на свим процесорима) $s(n)$, онда постоји статички EREW алгоритам са $T(n, s(n)/t(n)) = O(t(n))$.

Приметимо да ако је $s(n)$ једнако секвенцијалној сложености алгоритма, онда модификовани алгоритам има ефикасност $O(1)$.

Доказ. Нека је $T(n,p) \leq t(n)$ за све довољно велике n, и нека је a_i укупан број корака које извршавају сви процесори у i-том кораку алгоритма, $i = 1, 2, \ldots, t(n)$. Тада је $\sum_{i=1}^{t(n)} a_i = s(n)$. Ако је $a_i \leq s(n)/t(n)$, онда има довољно процесора за паралелно извршавање корака i. У противном се корак i замењује са $\lceil a_i/(s(n)/t(n)) \rceil$ корака у којима расположивих $s(n)/t(n)$ процесора емулирају кораке, које у оригиналном алгоритму извршава p процесора (користећи принцип имитирања паралелизма). Укупан број корака је сада

$$\sum_{i=1}^{t(n)} \left\lceil \frac{a_i}{s(n)/t(n)} \right\rceil \leq \sum_{i=1}^{t(n)} \left(\frac{a_i t(n)}{s(n)} + 1 \right) = t(n) + \frac{t(n)}{s(n)} \sum_{i=1}^{t(n)} a_i = 2t(n).$$

Према томе, време извршавања модификованог алгоритма је такође $O(t(n))$.

Ово тврђење познато је као Брентова лема. Брентова лема показује да је под одређеним претпоставкама ефикасност паралелног алгоритма одређена односом укупног броја операција (операција које извршавају сви процесори) и времена извршавања секвенцијалног алгоритма.

Ограничење да алгоритам буде статички је потребно, јер се мора знати које процесоре треба емулирати. Брентова лема је тачна и за алгоритме који нису статички, под условом да се емулација може лако извести. Пример где се ова лема не може применити је следећи. Претпоставимо да имамо n процесора и n елемената. После првог корака неки процесори одлучују (на основу резултата првог корака) да престану са радом. Исто се дешава и другом, трећем кораку, итд. Овај алгоритам је сличан турнирском алгоритму, изузев што се у овом случају не зна који процесори одустају од даљег рада. Ако покушамо да емулирамо преостале процесоре после на пример првог корака, потребно је да знамо који су још активни. Да би се то установило, потребно је извршити нека израчунавања.

Модел CRCW

Намеће се утисак да паралелни алгоритам не може да нађе максимум за мање од $\log_2 n$ корака, ако се користе само упоређивања. Међутим, то није тачно. Следећи алгоритам са временом извршавања $O(1)$ илуструје могућности истовремених уписа. Подразумева се варијанта истовремених уписа, у којој два или више процесора могу да пишу истовремено на исту локацију само ако записују исти податак. Због једноставности претпоставићемо да су сви елементи различити.

Користи се $n(n-1)/2$ процесора, тако да се процесор P_{ij} додељује пару елемената $\{x_i, x_j\}$. Поред тога, сваком елементу x_i придружује се

заједничка (дељена) променљива v_i, са почетном вредношћу 1. У првом кораку сваки процесор упоређује своја два елемента и записује 0 у променљиву придружену мањем елементу. Пошто је само један елеменат већи од свих осталих, само једна од променљивих v_i задржава вредност 1. У другом кораку процесори придружени победнику могу да установе да је он победник и да објаве ту чињеницу (на пример, уписивањем његовог индекса у посебну заједничку променљиву, резултат). Овај алгоритам захтева само два корака, независно од n. Међутим, његова ефикасност је врло мала, јер он захтева $O(n^2)$ процесора. Ово је такозвани *двокорачни алгоритам*.

Ефикасност двокорачног алгоритма може се побољшати као и алгоритма за модел EREW. Улазни подаци деле се у мале групе, тако да се свакој групи може доделити довољно процесора, да би се максимум групе могао одредити двокорачним алгоритмом. Са опадањем броја кандидата расте број расположивих процесора по кандидату, па се може повећати величина групе. Двокорачни алгоритам омогућује одређивање максимума у групи величине k са $k(k-1)/2$ процесора, за константно време. Претпоставимо да имамо укупно n процесора и да је n степен двојке. У првом циклусу величина сваке групе је 2 и максимум у свакој групи може се одредити у једном кораку. У други циклус улази се са $n/2$ елемената, и n процесора. Ако формирамо групе од по 4 елемента, имаћемо $n/8$ група, што нам омогућује да свакој групи доделимо 8 процесора. Ово је довољно, јер је $4 \cdot (4-1)/2 = 6$. У трећи циклус улази се са $n/8$ елемената. Покушајмо да одредимо највећу могућу величину групе која се може обрадити на овај начин. Ако је величина групе g, онда је број група $n/8g$, и за сваку групу имамо на располагању $8g$ процесора. За примену двокорачног алгоритма на групу величине g потребно је $g(g-1)/2$ процесора, па мора да буде $g(g-1)/2 \leq 8g$, односно $g \leq 17$; једноставније је узети вредност $g = 16$. Уопште, у i-ти циклус се улази са $n/2^{2^{i-1}-1}$ елемената, који се деле на $n/2^{2^i-1}$ група по $g = 2^{2^{i-1}}$ елемената, $i \geq 1$. За налажење максимума у групи двокорачним алгоритмом довољно је $g(g-1)/2 \leq g^2/2 = 2^{2^i-1}$ процесора, па је за налажење максимума у свим групама довољно

$$\frac{n}{2^{2^i-1}} \cdot 2^{2^i-1} = n$$

процесора. У наредни циклус улази се са по једним елементом из сваке групе, дакле са $n/2^{2^i-1}$ елемената, што индукцијом доказује исправност ове конструкције. Укупан број циклуса до завршетка алгоритма ограничен је условом да је број елемената на почетку i-тог циклуса мањи од један: $n/2^{2^{i-1}-1} \leq 1$, или $i \geq \log_2(\log_2 n + 1) + 1$. Дакле број циклуса, а тиме и број корака приликом извршења овог алгоритма је $O(\log \log n)$.

Иако је овај алгоритам нешто спорији од двокорачног ($O(\log \log n)$ у односу на $O(1)$), његова ефикасност је много боља. Она износи $O(1/\log \log n)$ у односу на $O(1/n)$ код двокорачног алгоритма. Описана техника може се назвати **подели и смрви** (енг. divide-and-crush), јер се улаз дели у групе, које су довољно мале да се могу "смрвити" мноштвом процесора. Примена ове технике није ограничена на модел CRCW.

10.3.3 Паралелни проблем префикса

Паралелни проблем префикса је важан јер се користи као основни елеменат при конструкцији многих паралелних алгоритама. Нека је $\star$ произвољна *асоцијативна* бинарна операција (операција која задовољава услов $x \star (y \star z) = (x \star y) \star z$) за произвољне x, y и z), коју ћемо означавати именом *производ*. На пример, $\star$ може да означава сабирање, множење или максимум два броја.

Проблем. Дат је низ бројева $x_1, x_2, \ldots, x_n$. Израчунати производе $x_1 \star x_2 \star \cdots \star x_k$ за $k = 1, 2, \ldots, n$.

Означимо са $PR(i, j)$ производ $x_i \star x_{i+1} \star \cdots \star x_j$. Потребно је израчунати $PR(1, k)$ за $k = 1, 2, \ldots, n$. Секвенцијална верзија проблема префикса је тривијална — префикси се једноставно израчунавају редом. Паралелни проблем префикса није тако лако решити. Искористићемо метод разлагања, уз уобичајену претпоставку да је n степен двојке.

Индуктивна хипотеза. Умемо да решимо паралелни проблем префикса за $n/2$ елемената.

Случај једног елемента је тривијалан. Алгоритам започиње поделом улаза на две половине, које се решавају индукцијом. На тај начин добијамо вредности $PR(1, k)$ и $PR(n/2{+}1, n/2{+}k)$ за $k = 1, 2, \ldots, n/2$. Прва половина ових вредности део је коначног резултата. Вредности $PR(1, m)$ за $m = n/2 + 1, n/2 + 2, \ldots, n$ добијају се израчунавањем производа $PR(1, n/2) \star PR(n/2{+}1, m)$. Оба ова члана позната су по индукцији (већ су израчуната; приметимо да је искоришћена асоцијативност операције $\star$). Алгоритам је приказан на слици 10.1. Чињеница да се проласци кроз **do** петљу извршавају паралелно (истовремено, на различитим скуповима процесора) у коду је назначена додатком "in parallel".

Сложеност. Улаз је подељен у два дисјунктна скупа у сваком рекурзивном позиву алгоритма. Оба потпроблема се могу дакле решити паралелно у моделу EREW. Ако имамо n процесора за проблем величине n, онда се половина њих може доделити сваком потпроблему. Комбиновање решења потпроблема састоји се од $n/2$ множења, која се могу извршити паралелно, али је потребан модел CREW, јер се у сваком множењу користи $PR(1, n/2)$, односно $x[Srednji]$. Иако више процесора истовремено читају $x[Srednji]$, они пишу на различите локације, па модел CRCW није неопходан (ако се алгоритам промени тако да сваки процесор има своју копију $x[Srednji]$). Укупан број корака је $T(n, n) = O(\log n)$, па је ефикасност алгоритма $E(n, n) = O(1/\log n)$ (време извршавања секвенцијалног алгоритма је $O(n)$).

На жалост, ефикасност овог алгоритма не може се побољшати коришћењем Брентове леме, јер је укупан број корака на свим процесорима $O(n \log n)$. Према томе, да би се побољшала ефикасност, мора се смањити укупан број корака.

Побољшање ефикасности паралелног префикса

Идеја која омогућује ефикасније решавање овог проблема је коришћење исте индуктивне хипотезе, али уз поделу улаза на другачији начин. Претпоставимо

Алгоритам Paralelni_Prefiks_1(x, n);
Улаз: x (низ са n елемената).
{претпоставља се да је n степен двојке}
Излаз: x (чији i-ти елеменат садржи i-ти префикс).
begin
    $PP\_1(1, n)$
end
procedure PP_1$(Levi, Desni)$;
begin
    if $Desni - Levi = 1$ **then**
        $x[Desni] := x[Levi] \star x[Desni]$ {$\star$ је асоцијативна бинарна операција}
    else
        $Srednji := (Levi + Desni - 1)/2$;
        do in parallel
            $PP\_1(Levi, Srednji)$; {додељено процесорима од 1 до $n/2$}
            $PP\_1(Srednji + 1, Desni)$; {додељено процесорима од $n/2 + 1$ до n}
        for $i := Srednji + 1$ **to** $Desni$ **do in parallel**
            $x[i] := x[Srednji] \star x[i]$
end

Слика 10.1: Алгоритам $Paralelni\_prefiks\_1$.

поново да је n степен двојке и да имамо n процесора. Нека E означава скуп свих x_i са парним индексима i. Ако израчунамо префиксе свих елемената из E, онда је израчунавање осталих префикса (оних са непарним индексима) лако: ако је познато $PR(1, 2i)$, онда се непарни префикс $PR(1, 2i + 1)$ добија израчунавањем само још једног производа $PR(1, 2i) \star x_{2i+1}$, $i = 1, 2, \ldots, n/2 - 1$, Префикси елемената из E могу се одредити у две фазе. Најпре се (паралелно) израчунавају производи $x_{2i-1} \star x_{2i}$, који се затим смештају у x_{2i}, $i = 1, 2, \ldots, n/2$. Другим речима, израчунавају се производи свих елемената из E са својим левим суседима. Затим се решава (индукцијом) проблем паралелног префикса за $n/2$ елемената из E. Резултат за свако x_{2i} је тачан коначни префикс, јер је свако x_{2i} већ замењено производом са x_{2i-1}. Пошто се знају префикси за све елементе са парним индексима, преостали префикси се могу израчунати у једном паралелном кораку на већ споменути начин. Лако се проверава да се овај алгоритам може извршавати у моделу EREW. Алгоритам је приказан на слици 10.2.

Сложеност. Обе петље у алгоритму $Paralelni\_prefiks\_2$ могу се извршити паралелно за време $O(1)$ са $n/2$ процесора. Рекурзивни позив примењује се на проблем двоструко мање величине, па је време извршавања алгоритма $O(\log n)$. Укупан број корака $s(n)$ задовољава диференцну једначину $s(n) = s(n/2) + n - 1$, $s(2) = 1$, из чега следи да је $s(n) = O(n)$ (прецизније, $s(2^k) = 2^{k+1} - k - 2$). Због тога се сада може искористити Брентова лема за побољшање ефикасности: алгоритам се може променити тако да се за време $O(\log n)$ извршава на само $O(n/\log n)$ процесора, односно да му ефикасност буде $O(1)$. Кључна идеја побољшања је коришћење само једног рекурзивног позива (уместо два), при чему се корак обједињавања и даље

```
Алгоритам Paralelni_Prefiks_2(x, n);
Улаз: x (низ са n елемената).
 {претпоставља се да је n степен двојке}
Излаз: x (чији i-ти елеменат садржи i-ти префикс).
begin
    PP_2(1)
end
procedure PP_2(Korak);
begin
    if Korak = n/2 then
        x[n] := x[n/2] ⋆ x[n] {⋆ је асоцијативна бинарна операција}
    else
        for i := 1 to n/(2 · Korak) do in parallel
            x[2 · i · Korak] := x[(2 · i − 1) · Korak] ⋆ x[2 · i · Korak];
        PP_2(2 · Korak);
        for i := 1 to n/(2 · Korak) − 1 do in parallel
            x[(2 · i + 1) · Korak] := x[2 · i · Korak] ⋆ x[(2 · i + 1) · Korak]
end
```

Слика 10.2: Алгоритам $Paralelni\_prefiks\_2$.

извршава паралелно.

10.3.4 Одређивање рангова у повезаној листи

У паралелним алгоритмима много је теже радити са повезаним листама него са низовима, јер су листе суштински секвенцијалне. Повезаној листи може се приступити само преко главе (првог елемента), и листа се мора пролазити елемент по елемент, без могућности паралелизације. У многим случајевима су, међутим, елементи листе (односно показивачи на њих) смештени у низ; редослед елемената листе независан је од редоследа у низу. У таквим случајевима, кад се листи приступа паралелно, постоји могућност примене брзих паралелних алгоритама.

Ранг елемента у повезаној листи дефинише се као растојање елемента од краја листе. Тако, на пример, први елеменат има ранг n, други $n-1$, итд.

Проблем. Дата је повезана листа од n елемената који су смештени у низ A дужине n. Израчунати рангове свих елемената листе.

Секвенцијални проблем се може решити простим проласком кроз листу. Метод који ћемо искористити за конструкцију паралелног алгоритма зове се **удвостручавање**. Сваком елементу додељује се по један процесор. На почетку сваки процесор зна само адресу десног (наредног) суседа свог елемента у листи. После првог корака сваки процесор зна елеменат на растојању два (дуж листе) од свог елемента. Ако у кораку i сваки процесор зна адресу елемента на растојању k од свог елемента, онда у наредном кораку сваки процесор може да пронађе адресу елемента на растојању $2k$.

Процес се наставља све док сви процесори не достигну крај листе. Нека је $N[i]$ адреса елемента десно од елемента i у листи, коју зна процесор P_i. На почетку је $N[i]$ десни сусед елемента i (изузев за последњи елеменат у листи, чији је показивач на десног суседа **nil**). У суштини, процесор P_i у сваком кораку замењује $N[i]$ вредношћу $N[N[i]]$, све док не достигне крај листе. Нека је $R[i]$ ранг елемента i. На почетку се променљивој $R[i]$ додељује вредност 0, изузев за последњи елеменат у листи, за кога се она поставља на вредност 1 (овај елеменат се од осталих разликује по показивачу, који има вредност **nil**). Кад процесор добије адресу суседа са рангом R различитим од нуле, он може да израчуна свој ранг (односно ранг свог елемента). На почетку само елеменат ранга 1 зна свој ранг. После првог корака елеменат ранга 2 открива да његов сусед има ранг 1, па закључује да је његов сопствени ранг 2. После другог корака елементи са рангом 3 и 4 установљавају своје рангове, итд. Ако P_i установи да $N[i]$ показује на "рангирани" елеменат ранга R после d корака удвостручавања, онда је ранг елемента i једнак $2^{d-1}+R$. Овај алгоритам (слика 10.3) се може лако прилагодити моделу EREW, (довољно је да сваки процесор независно израчунава своју копију $D[i]$ променљиве D).

```
Алгоритам Rangovi(N);
Улаз: N (низ од n елемената).
Излаз: R (рангови свих елемената у низу).
begin
    D := 1;
    {Сваки процесор може имати своју локалну променљиву D}
    {овде је D заједничка променљива}
    do in parallel {процесор P_i је активан док R[i] не постане различито од нуле}
        R[i] := 0;
        if N[i] = nil then R[i] := 1;
        while R[i] = 0 do
            if R[N[i]] ≠ 0 then
                R[i] := D + R[N[i]]
            else
                N[i] := N[N[i]];
                D := 2 · D
end
```

Слика 10.3: Паралелни алгоритам за одређивање рангова елемената повезане листе.

Сложеност. Процес удвостручавања омогућује да сваки процесор достигне крај листе после највише $\lceil \log_2 n \rceil$ корака, па је $T(n,n) = O(\log n)$. Ефикасност алгоритма је $E(n,n) = O(1/\log n)$. Поправка ефикасности захтевала би темељну прераду алгоритма, јер је укупан број корака $O(n \log n)$.

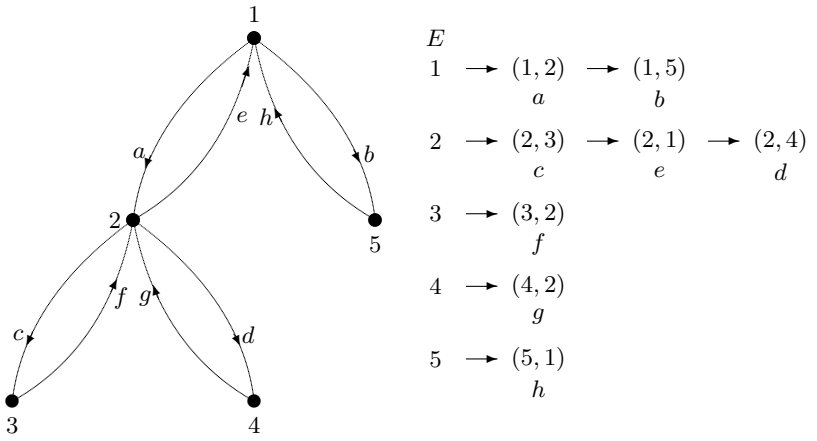
Познавање рангова омогућује трансформацију листе у низ за $O(\log n)$ паралелних корака. После израчунавања свих рангова, елементи се могу паралелно прекопирати на своје локације у низу, па се остатак израчунавања

може извршити директно на низу, после чега је њихова обрада много једноставнија.

10.3.5 Техника Ојлеровог обиласка

Многи алгоритми за рад са стаблима могу се паралелизовати тако да се паралелно обрађује комплетна генерација чворова (на пример, код турнирског алгоритма за налажење максимума). Време извршавања таквог алгоритма је пропорционално висини стабла. Ако је стабло са n чворова довољно уравнотежено и висина му је $O(\log n)$, онда је овај приступ сасвим добар. Међутим, ако стабло није уравнотежено, висина стабла може да буде у најгорем случају $n-1$, па се мора тражити неки други приступ. *Техника Ојлеровог обиласка* је алатка за конструкцију паралелних алгоритама на стаблима, погодна и за неуравнотежена стабла.

Нека је T стабло. Претпоставимо да је T представљено уобичајеном листом повезаности, уз једну допуну. Као и обично, постоји показивач $E[i]$ на почетак листе грана суседних чвору i (ако је ова листа празна, онда $E[i]$ има вредност **nil**). Та листа састоји се од слогова који садрже одговарајућу грану (i,j) (при томе је довољно сместити само j, јер је i познато) и показивач $Naredna(i,j)$ на наредну грану у листи. Свака неусмерена грана (i,j) представљена је са две усмерене копије, (i,j) и (j,i). Слогови листе садрже и додатни показивач: слог који одговара грани (i,j) садржи показивач на грану (j,i). Ово је потребно да би се грана (j,i) могла брзо пронаћи кад се зна адреса гране (i,j). Пример овако представљеног стабла дат је на слици 10.4; показивачи на копије грана због прегледности нису приказани.



Слика 10.4: Репрезентација стабла.

Техника Ојлеровог обиласка заснива се на идеји да се формира листа грана стабла, и то оним редом којим се гране појављују у Ојлеровом циклусу за усмерену верзију стабла (у циклусу се свака неусмерена грана појављује два пута, по једном у оба смера). Кад се зна ова листа, многе операције са стаблом могу се извести директно на листи, као да је листа линеарна. Секвенцијалним алгоритмом лако се може прегледати стабло и успут извршити

потребне операције. Оваква "линеаризација" омогућује да се операције са стаблом ефикасно изводе паралелно. Видећемо два примера таквих операција, пошто претходно размотримо формирање Ојлеровог циклуса.

Секвенцијално налажење Ојлеровог обиласка стабла T (у коме се свака грана појављује два пута) је једноставно. Може се извести обилазак стабла користећи претрагу у дубину, враћајући се супротно усмереном граном приликом сваког повратка назад у току претраге. Слична ствар се може извести и паралелно. Нека $w(i,j)$ означава грану која следи иза гране (i,j) у циклусу. Испоставља се да се $w(i,j)$ може дефинисати следећом једнакошћу

$$w(i,j) = \begin{cases} Naredna(j,i) & \text{ако } Naredna(j,i) \text{ није } \textbf{nil} \\ E[j] & \text{у осталим случајевима} \end{cases},$$

на основу које се лако паралелно израчунава. Другим речима, листа грана суседних чвору j пролази се цикличким редоследом (ако је (j,i) последња грана у листи чвора j, онда се узима прва грана из те листе, она на коју показује $E[j]$). На пример, ако кренемо од гране a на слици 10.4, онда се циклус састоји од грана a, d, g, c, f, e, b, h, и поново a. Чињеница да $Naredna(j,i)$ следи иза (i,j) у циклусу обезбеђује да ће грана (j,i) доћи на ред после проласка свих грана суседних чвору j. Према томе, подстабло са кореном у j биће комплетно прегледано пре повратка у чвор i.

Кад је листа грана у Ојлеровом обиласку конструисана, произвољна грана (r,t) бира се за полазну, а грана која јој претходи означава се као крај листе. Чвор r се бира за *корен* стабла. После тога се гране алгоритмом *Rangovi* са слике 10.3 могу нумерисати, у складу са својим положајем у листи. Нека $R(i,j)$ означава ранг гране (i,j) у листи. Тако је, на пример, $R(r,t) = 2(n-1)$, где је n број чворова. Приказаћемо сада два примера операција са стаблом — долазну нумерацију чворова, и израчунавање броја потомака за све чворове.

За грану (i,j) у циклусу кажемо да је *директна грана* ако је усмерена од корена, односно да је *повратна грана* у противном. Нумерација чворова омогућује разликовање директних од повратних грана: грана (i,j) је директна грана ако и само ако је $R(i,j) > R(j,i)$. Пошто су две копије гране (i,j) повезане показивачима, лако је установити која је од њих директна грана. Шта више, ова провера се може обавити паралелно за све гране. Директне гране су интересантне, јер одређују редослед чворова при долазној нумерацији. Нека је (i,j) директна грана која води до чвора j (односно, чвор i је отац чвора j у стаблу). Ако је $f(i,j)$ број директних грана које следе иза (i,j) у листи, онда је редни број чвора j једнак $n - f(i,j)$. Редни број корена r, јединог чвора до кога не води ни једна директна грана, је 1. Применом варијанте алгоритма са удвостручавањем може се израчунати вредност $f(i,j)$ за сваку директну грану (i,j). Прецизнију разраду алгоритма остављамо читаоцу као вежбање.

Други пример је израчунавање броја потомака сваког чвора у стаблу. Нека је (i,j) (јединствена) директна грана која води до задатог чвора j. Посматрајмо гране које следе иза гране (i,j) у листи. Број чворова испод j у стаблу једнак је броју директних грана испод j у стаблу. Ми већ знамо како се паралелно израчунавају вредности $f(i,j)$, једнаке броју директних грана које следе иза гране (i,j) у листи. На сличан начин $f(j,i)$ је број

директних грана које следе иза гране (j,i) у листи. Лако је видети да је број потомака чвора j једнак $f(i,j)-f(j,i)$. Време извршавања оба описана алгоритма на моделу EREW је $T(n,n)=O(\log n)$

10.4 Алгоритми за мреже рачунара

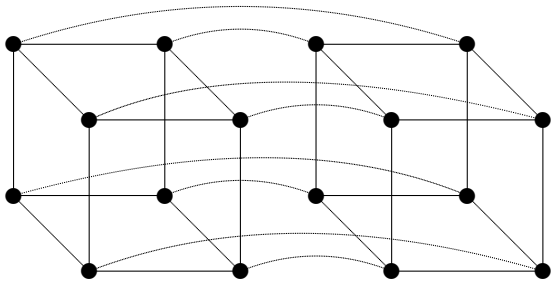
Мреже рачунара могу се моделирати графовима, обично неусмереним. Процесори одговарају чворовима, а два чвора су повезана граном ако постоји директна веза између одговарајућих процесора. Сваки процесор има своју локалну меморију, а посредством мреже може да приступи локалним меморијама других процесора. Према томе, сва меморија је на неки начин заједничка, али цена приступа некој променљивој зависи од локација процесора и променљиве. Приступ жељеној променљивој може бити брз колико и локални приступ (ако је променљива у истом процесору), или толико спор колико и пролазак кроз целу мрежу (у случају кад граф има облик низа повезаних чворова). Трајање приступа је негде између ове две крајности. Процесори комуницирају разменом порука. Кад процесор треба да приступи променљивој смештеној у локалној меморији другог процесора, он шаље поруку са захтевом за променљивом. Порука се усмерава кроз мрежу.

Више различитих графова користе се за мреже рачунара. Најједноставнији међу њима су линеарни низ, прстен, бинарно стабло, звезда и дводимензионална мрежа. Ефикасност комуникације расте са бројем грана у мрежи. Међутим, гране су скупе — то се може објаснити повећањем површине коју заузимају везе, а тиме повећањем димензија мреже и времена комуникације. Због тога се обично тражи компромис. Не постоји тип графа који је универзално добар. Погодност одређеног графа битно зависи од начина комуницирања у оквиру конкретног алгоритма. Међутим, постоје неке особине графова, које могу бити врло корисне за више различитих алгоритама. У наставку ћемо их навести, као и примере мрежа рачунара.

Битан параметар мреже је **дијаметар** одговарајућег графа, тј. највеће растојање нека два чвора. Дијаметар одређује максимални број грана на путу порука до одредишта. Дводимензионална мрежа $n\times n$ има дијаметар $2n$, а комплетно бинарно стабло са n чворова има дијаметар $2\log_2(n+1)-2$. према томе може да испоручи поруку много брже него дводимензионална мрежа. С друге стране, стабло има уско грло, јер сав саобраћај из једне у другу половину стабла пролази кроз корен. Дводимензионална мрежа нема уско грло и врло је симетрична, што је важно за алгоритме у којима је комуникација симетрична.

Хиперкоцка је популарна структура која комбинује предности високе симетрије, малог дијаметра, мноштва алтернативних путева између два чвора и одсуства уских грла. d-димензионална хиперкоцка састоји се од $n=2^d$ процесора. Адресе процесора су d-торке бројева из скупа $\{0,1\}$ (које се могу кодирати бројевима од 0 до 2^d-1). Према томе, свака адреса се састоји од d бита. Процесор P_i је повезан са процесором P_j ако и само ако се бинарни запис i разликује од бинарног записа j на тачно једном биту. Растојање између произвољна два процесора је увек мање или једнако од d, јер се од P_i до P_j може доћи променом највише d бита, једног по једног. На слици 10.5 приказана је четвородимензионална хиперкоцка. Хиперкоцка обезбеђује богатство веза, јер постоји много различитих путева између

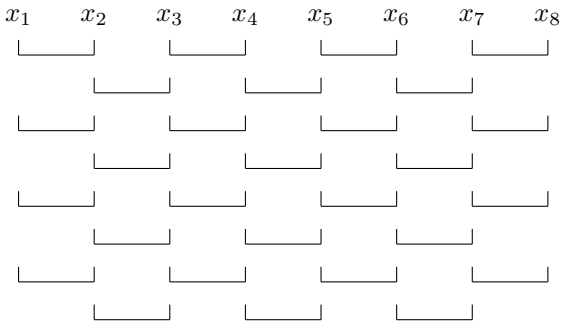
свака два процесора (одговарајући бити могу се мењати произвољним редоследом). Хиперкоцка се може такође комбиновати са неком другом мрежом, на пример умећући мреже уместо темена хиперкоцке. У примени се појављују и друге структуре мрежа.



Слика 10.5: Четвородимензионална хиперкоцка.

10.4.1 Сортирање на низу

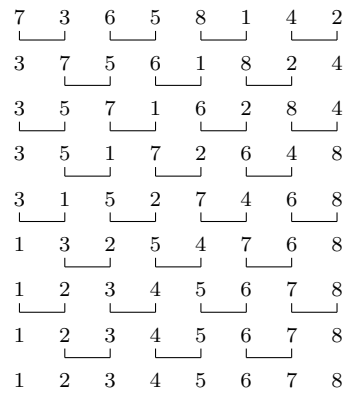
Размотрићемо најпре једноставан проблем сортирања на низу процесора. На располагању је n процесора $P_1, P_2, \ldots, P_n$ и задато је n бројева $x_1, x_2, \ldots, x_n$. Сваки процесор чува један улазни податак. Циљ је прерасподелити бројеве међу процесорима тако да најмањи од њих буде у P_1, следећи у P_2, итд. У општем случају могуће је додељивање више улазних података једном процесору. Видећемо да се алгоритам може прилагодити и таквим условима. Процесори су повезани у линеарни низ: процесор P_i је повезан са процесором P_{i+1}, $i = 1, 2, \ldots, n-1$. Пошто сваки процесор може да комуницира само са суседима, упоређивање и размену података могуће је вршити само између елемената који су суседни у низу. У најгорем случају алгоритам захтева извршавање $n - 1$ корака, колико је потребно да се податак премести са једног на други крај низа. Алгоритам се у основи извршава на следећи начин. Сваки процесор упоређује свој број са бројем једног од својих суседа, размењује бројеве ако је њихов редослед погрешан, а затим исти посао обавља са другим суседом (суседи се морају смењивати, јер би се у противном упоређивали увек исти бројеви). Исти процес наставља се све док бројеви не буду поређани на жељени начин. Кораци се деле на *непарне* и *парне*. У непарним корацима процесори са непарним индексом упоређују своје са бројевима својих десних суседа; у парним корацима процесори са парним индексом упоређују своје са бројевима својих десних суседа (слика 10.6). На тај начин су сви процесори синхронизовани и упоређивање увек врше процесори који то и треба да раде. Ако процесор нема одговарајућег суседа (на пример први процесор у другом кораку), он у току тог корака мирује. Овај алгоритам се зове **сортирање парно-непарним транспозицијама**, видети слику 10.7. Пример рада алгоритма приказан је на слици 10.8. Приметимо да се у овом примеру сортирање завршава после само шест корака. Ипак, ранији завршетак тешко је открити у мрежи. Према томе, боље је оставити алгоритам да се извршава до свог завршетка у најгорем случају.



Слика 10.6: Сортирање парно-непарним транспозицијама.

Алгоритам Sortiranje_na_nizu(x, n);
Улаз: x (низ са n елемената, при чему је x_i у процесору P_i).
Излаз: x (сортирани низ, тако да је i-ти најмањи елеменат у P_i).
begin
  do in parallel $\lceil n/2 \rceil$ пута
    P_{2i-1} и P_{2i} упоређују своје елементе и по потреби их размењују;
      {за све i, такве да је $1 < 2i \leq n$}
    P_{2i} и P_{2i+1} упоређују своје елементе и по потреби их размењују;
      {за све i, такве да је $1 \leq 2i < n$}
      {ако је n непарно, овај корак се извршава само $\lfloor n/2 \rfloor$ пута}
end

Слика 10.7: Алгоритам за сортирање на низу процесора.



Слика 10.8: Пример сортирања парно-непарним транспозицијама.

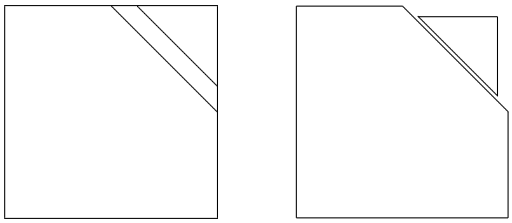
Алгоритам *Sortiranje_na_nizu* изгледа природно и јасно, али доказ исправности његовог рада није тривијалан. На пример, елемената се у неким корацима може *удаљавати* од свог коначног положаја. У примеру на слици 10.8 број 5 се креће улево два корака пре него што почне са кретањем удесно, а 3 иде до левог краја и остаје тамо три корака пре него што крене назад удесно. Доказ исправности рада паралелних алгоритама није једноставан, због међузависности деловања различитих процесора. Понашање једног процесора утиче на све остале процесоре, па је обично тешко усредсредити се на један процесор и доказати да је то што он ради исправно; морају се посматрати сви процесори заједно.

Теорема 15. *На крају извршавања алгоритма Sortiranje_na_nizu дати бројеви су сортирани.*

Доказ. Доказ се изводи индукцијом по броју елемената. При томе се тврђење мало појачава: сортирање се завршава после n корака, без обзира да ли је први корак паран или непаран. За $n = 2$ сортирање се завршава после највише два корака; сортирање траје два корака ако је први корак непаран. Претпоставимо да је теорема тачна за n процесора, и посматрајмо случај $n+1$ процесора. Сконцентришимо пажњу на максимални елеменат, и претпоставимо да је то x_m (у примеру на слици 10.8 то је x_5). У првом кораку x_m се упоређује са x_{m-1} или x_{m+1}, зависно од тога да ли је m парно или непарно. Ако је m парно, нема замене јер је x_m веће од x_{m-1}. Исход је исти као у случају да је број x_m на почетку био у процесору P_{m-1}, и да је замена била извршена. Према томе, без губитка општости може се претпоставити да је m непарно. У том случају број x_m се упоређује са x_{m+1}, замењује, и као највећи се премешта корак по корак удесно (дијагонално на слици 10.8), све док не дође на место x_{n+1}, а онда остаје тамо. То је позиција коју x_m и треба да заузме, па сортирање исправно третира максимални елемент.

Показаћемо сада индукцијом да је и сортирање осталих n елемената коректно. Посматрајмо дијагоналу насталу кретањем максималног елемената (видети слику 10.9). Упоређивања у којима учествује максимални елеменат се игноришу. Упоређивања делимо на две групе, она испод, и она изнад дијагонале. Затим "транслирамо" троугао изнад дијагонале за једно поље наниже и једно поље улево. Другим речима, за упоређивања у горњем троуглу корак i се сада зове $i+1$. На пример, посматрајмо упоређивања 1 са 8 и 4 са 2 у првом кораку на слици 10.8. Прво упоређивање је на дијагонали, па се игнорише; друго је изнад дијагонале, па се сматра делом корака 2, уместо корака 1. Према томе, корак 2 састоји се од упоређивања 7 са 5, 6 са 1, и 4 са 2. Међутим, ово је регуларни парни корак са само n елемената. Корак 1 и сва упоређивања у којима учествује максимални елеменат се сада могу просто игнорисати, после чега су преостала упоређивања идентична са низом упоређивања која се изводе при извршавању алгоритма са n елемената (при чему је први корак непаран). Према индуктивној хипотези сортирање n елемената се изводи коректно; према томе, и сортирање свих $n+1$ елемената је коректно, а завршава се после $n+1$ корака.

До сада смо разматрали случај једног елемента по процесору. Претпоставимо сада да сваки процесор памти k елемената, и размотримо за почетак случај само два процесора. Претпоставимо да је циљ да процесори



Слика 10.9: Корак индукције у доказу исправности сортирања парно-непарним транспозицијама, Теорема 15.

међусобно размене елементе, тако да k најмањих елемената дођу у P_1, а k највећих у P_2. Јасно је да у најгорем случају сви елементи морају бити размењени, па је тада број размена елемената $2k$. Сортирање се може извести понављањем следећег корака све док је потребно: P_1 шаље свој највећи елеменат у P_2, а P_2 свој најмањи елеменат у P_1. Процес се завршава у тренутку кад је највећи елеменат у P_1 мањи или једнак од најмањег елемента у P_2. Овај корак зове се *обједињавање-раздвајање*. Користећи овај корак као основни у сортирању парно-непарним транспозицијама, добија се уопштење сортирања на случај више елемената по процесору.

Иако је овакав алгоритам сортирања оптималан за низ процесора, његова ефикасност је мала. Имамо n процесора који извршавају n корака; према томе, укупан број корака је n^2. Мала ефикасност $O(\log n/n)$ није изненађујућа, јер ефикасан алгоритам за сортирање мора имати могућност да замењује места међусобно удаљеним елементима. Низ процесора не омогућује такве замене. У следећем одељку приказаћемо специјализоване мреже за ефикасно сортирање.

10.4.2 Мреже за сортирање

Кад конструишемо ефикасан секвенцијални алгоритам, интересује нас само укупан број корака. У случају конструкције паралелних алгоритама, циљ је учинити кораке што независнијим. Посматрајмо сортирање обједињавањем. Два рекурзивна позива су потпуно независна и могу се извршити паралелно. Међутим, обједињавање као део алгоритма извршава се на секвенцијални начин. У излазни низ се i-ти елеменат ставља тек кад је тамо већ стављено првих $i-1$ елемената. Ако нам пође за руком паралелизација обједињавања, онда ћемо моћи да паралелизујемо и сортирање обједињавањем.

Описаћемо сада другачији алгоритам обједињавања, заснован на техници разлагања. Претпоставимо због једноставности да је n степен двојке. Нека су $a_1, a_2, \ldots, a_n$; $b_1, b_2, \ldots, b_n$ два сортирана низа које треба објединити, и нека је $x_1, x_2, \ldots, x_{2n}$ резултат њиховог обједињавања. Специјално је, на пример, $x_1 = \min\{a_1, b_1\}$ и $x_{2n} = \max\{a_n, b_n\}$. Потребно је објединити различите делове ових низова паралелно, тако да њихово комплетно обједињавање буде једноставно. Циљ се може постићи поделом два низа на по два дела — са непарним, односно парним индексима. Сваки део се обједињава са одговарајућим делом другог низа, а онда се комплетира обједињавање. Нека је $o_1, o_2, \ldots, o_n$ обједињени редослед непарних низова $a_1, a_3, \ldots, a_{n-1}$; b_1, b_3, …, b_{n-1}, и нека је $e_1, e_2, \ldots, e_n$ обједињени редослед парних низова

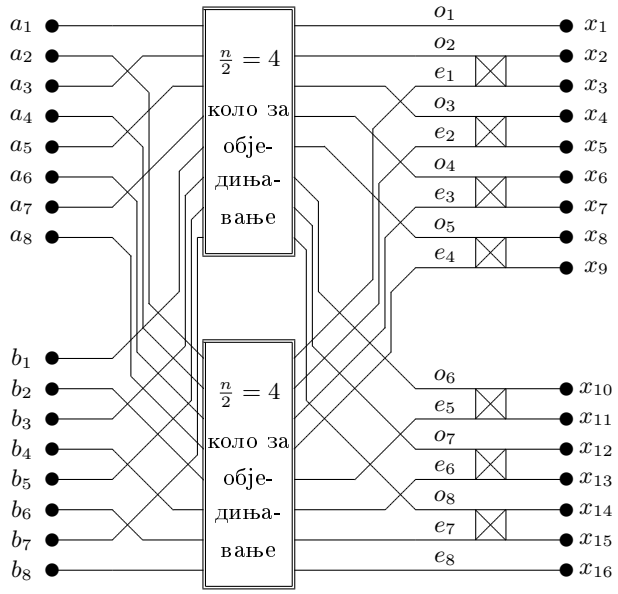
$a_2, a_4, \ldots, a_n$; b_2, b_4, $\ldots$, b_n. Очигледно је $x_1 = o_1$ и $x_{2n} = e_n$. Наредна теорема показује да се и остатак обједињавања такође лако изводи.

Теорема 16. *У складу са уведеним ознакама, за* $i = 1, 2, \ldots, n-1$ *важи* $x_{2i} = \min\{o_{i+1}, e_i\}$ *и* $x_{2i+1} = \max\{o_{i+1}, e_i\}$.

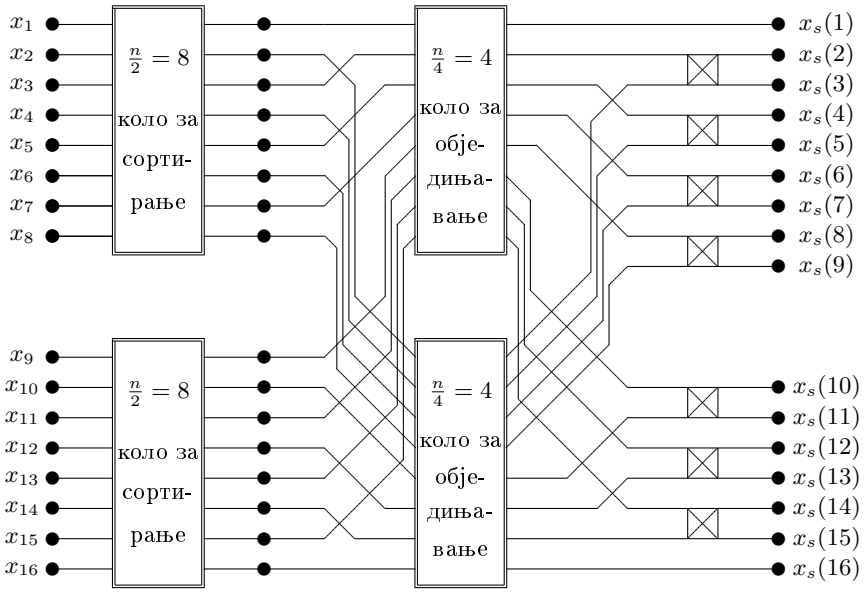
Доказ. Због једноставности се под парним, односно непарним елементима подразумевају елементи низова a_i, b_i са парним, односно непарним индексом. Размотримо елеменат e_i. Пошто је e_i i-ти елеменат у сортираном редоследу парних низова, e_i је веће или једнако од *бар* i парних елемената у оба низа. Међутим, сваком парном елементу који је мањи или једнак од e_i одговара још један непарни елеменат који је такође мањи или једнак од e_i (јер смо пошли од два сортирана низа). Према томе, e_i је веће или једнако од бар $2i$ елемената из оба низа. Другим речима, установили смо да је $e_i \geq x_{2i}$. На сличан начин, o_{i+1} је веће или једнако од бар $i+1$ непарних елемената, из чега следи да је веће или једнако од бар $2i$ елемената укупно (за сваки непаран елеменат *сем првог*, који је мањи или једнак од o_{i+1}, постоји још један парни елеменат који је мањи или једнак од o_{i+1}). Према томе, важи и неједнакост $o_{i+1} \geq x_{2i}$. Специјално, за $i = n-1$ добија се $e_{n-1} \geq x_{2n-2}$ и $o_n \geq x_{2n-2}$, па пошто је $e_n = x_{2n}$, тврђење теореме је тачно за $i = n-1$. Стављајући даље у горње неједнакости редом $i = n-2, n-3, \ldots, 1$, добијамо да је тврђење теореме тачно и за остале вредности i, чиме је завршен доказ теореме.

Важна последица Теореме 16 је да се комплетно обједињавање низова $o_1, o_2, \ldots, o_n$ и $e_1, e_2, \ldots, e_n$ може извршити у једном паралелном кораку. Остатак посла извршавају рекурзивни позиви описаног алгоритма. Конструкција паралелног алгоритма следи директно из теореме. Слика 10.10 илуструје рекурзивну конструкцију обједињавања, а на слици 10.11 се види пример комплетног сортирања, које се зове **сортирање парно-непарним обједињавањем**. Правоугаоници на левој страни слике 10.11, означени са "$n/2$ сортирање", представљају рекурзивне копије комплетне мреже за сортирање, које сортирају по $n/2$ бројева.

Сложеност. Диференцна једначина за укупан број корака $T_M(n)$ за обједињавање је $T_M(2n) = 2T_M(n) + n - 1$, $T_M(1) = 1$. Из тога следи да је укупан број упоређивања $O(n \log n)$, у поређењу са секвенцијалним алгоритмом који захтева само $O(n)$ корака. Дубина рекурзије, која одговара броју паралелних корака, је $O(\log n)$. Диференцна једначина за укупан број корака $T_S(n)$ за сортирање парно–непарним обједињавањем је $T_S(2n) = 2T_S(n) + O(n \log n)$, $T_S(2) = 1$. Њено решење је $T_S(n) = O(n \log^2 n)$. Мрежа садржи по n процесора у свакој "колони", а њена дубина (тј. број колона) је $O(\log^2 n)$, па је укупан број процесора у мрежи $O(n \log^2 n)$. Приметимо да би се истих n процесора могли користити у свим колонама, али они би тада морали бити скоро потпуно исповезивани. Једини тип израчунавања у мрежи је упоређивање и евентуална замена, па су потребни једноставни процесори који се састоје од компаратора са два улаза и два излаза.



Слика 10.10: Коло за парно–непарно обједињавање два низа од по $n = 8$ елемената.



Слика 10.11: Пример сортирања 16 бројева помоћу кола за сортирање парно-непарним обједињавањем.

10.4.3 Налажење k-тог најмањег елемента на стаблу

Претпоставимо сада да је мрежа рачунара комплетно бинарно стабло висине $h-1$ са $n=2^{h-1}$ листова, односно 2^h-1 процесора, придружених чворовима стабла. Улаз је низ $x_1, x_2, \ldots, x_n$, при чему се у почетном тренутку x_i чува у листу i. Рачунари у облику стабла користе се нпр. у вези са обрадом слика, при чему листови одговарају елементима улазног низа а алгоритми који их обрађују су хијерархијски. Овде ћемо размотрити проблем налажења k-тог најмањег елемента.

Подсетимо се најпре секвенцијалног алгоритма за налажење k-тог најмањег елемента. Због једноставности претпоставимо да су елементи низа различити. Алгоритам је пробабилистички. У сваком кораку бира се случајни елеменат x као **пивот**. Ранг елемента x израчунава се упоређивањем x са свим осталим елементима, а онда се елиминишу елементи који су мањи или већи од x, зависно од тога да ли је ранг мањи или већи од k. Извршавање алгоритма обуставља се у тренутку кад је ранг пивота k. Очекивани број итерација је $O(\log n)$, а очекивани број упоређивања је $O(n)$. Постоје три различите фазе у свакој итерацији алгоритма: (1) избор случајног елемента, (2) израчунавање његовог ранга, и (3) елиминација. Описаћамо ефикасну паралелну реализацију сваке од фаза.

Избор случајног елемента може се постићи организовањем турнира на стаблу. Сваки лист шаље свој број оцу, где се он "такмичи" са бројем брата. Победник у партији одређује се бацањем новчића. Број који је победио прелази у друго коло такмичења — иде увис по стаблу. Процес се наставља све дотле док корен стабла не изабере тачно један број као укупног победника (овај поступак је регуларан само у првој итерацији; размотрићемо касније како га треба поправити, да би радио исправно и кад су неки елементи елиминисани). Број–победник се затим шаље низ стабло до свих листова, ниво по ниво; у процесу слања учествује сваки чвор стабла, шаљући добијени број и левом и десном сину. Пошто сви листови сазнају који је број пивот, они своје бројеве могу да упореде са пивотом у једном паралелном кораку. Листови затим шаљу навише, свом оцу, јединицу, ако је њихов број мањи или једнак од пивота, односно нулу у противном. Ранг пивота је број јединица послатих навише. Сабирање бројева који се шаљу навише лако се изводи тако што сваки чвор сабира бројеве добијене од синова. После тога корен шаље наниже ранг пивота, и сваки лист може независно да установи да ли треба елиминише свој број. Укупно постоје четири "таласа" комуникације у свакој итерацији: (1) уз стабло да би се изабрао пивот, (2) низ стабло, да би се пивот послао до листова, (3) уз стабло, да би се израчунао ранг пивота, и (4) низ стабло се шаље ранг пивота.

После прве итерације појављује се један проблем; пошто су неки елементи елиминисани, такмичари више нису равноправни. Посматрајмо, на пример, екстремни случај кад су у једној половини стабла елиминисани сви листови сем једног. Преостали елеменат у тој половини стабла "провући" ће се до "финала" без икаквог такмичења, а онда ће бити изабран са вероватноћом $1/2$, док је вероватноћа избора осталих елемената много мања. Волели бисмо да очувамо униформност расподеле вероватноћа приликом избора пивота у свакој итерацији. Униформност се може очувати на следећи начин. Процесори, чији су бројеви елиминисани у претходним колима, шаљу навише

специјалну вредност **nil**, коју сваки елеменат побеђује. Сваком елементу који учествује у такмичењу придружује се бројач, чија је почетна вредност 1. Бројач показује колико је стварних "противника" учествовало у делу турнира, из кога је елеменат изашао као победник (односно број неелиминисаних елемената у подстаблу чвора, у коме се налази елеменат). Кад елеменат победи у партији у неком чвору стабла, он се упућује навише, а вредност његовог бројача повећава се за вредност бројача његовог пораженог противника (као у филму "Горштак"). Партије се одигравају са "несиметричним" новчићем, код кога је однос вероватноћа нуле и јединице једнак односу вредности бројача два такмичара. Ако су бројачи играча p, односно q, онда први играч побеђује са вероватноћом $p/(p+q)$, а други са вероватноћом $q/(p+q)$. На пример, ако је у првој партији x победио играча y, а z прошао даље без борбе, онда бројачи елемената x, односно z имају вредности 2, односно 1. У игри x против z користи се новчић који је наклоњен елементу x, односно додељује му предност у односу $2:1$. Резултат је да z има вероватноћу $1/3$ победе у овој партији, а x и y са вероватноћом $(1/2)\cdot(2/3) = 1/3$ побеђују у обе своје партије. Индукцијом се показује да овакав процес избора обезбеђује да у сваком колу сви елементи се једнаком вероватноћом могу бити изабрани за пивот.

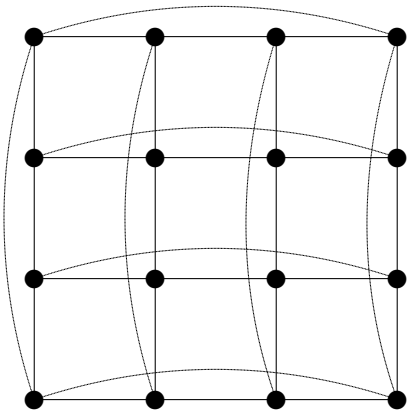
Сложеност. Број паралелних корака у свакој итерацији једнак је четворострукој висини стабла. Пошто овај алгоритам елиминише елементе на потпуно исти начин као и секвенцијални алгоритам, очекивани број итерација је $O(\log n)$. Према томе, очекивано време извршавања је $O(\log^2 n)$.

10.4.4 Множење матрица на дводимензионалној мрежи

Мрежа рачунара коју ћемо сада размотрити је дводимензионална мрежа $n\times n$. Процесор $P[i,j]$ налази се у пресеку i-те врсте и j-те колоне, и повезан је са процесорима $P[i,j+1]$, $P[i+1,j]$, $P[i,j-1]$ и $P[i-1,j]$. Претпоставља се да су супротни крајеви мреже међусобно повезани, односно да мрежа личи на торус. Тако, на пример, $P[0,0]$ је повезан са $P[0,n-1]$ и $P[n-1,0]$, поред $P[0,1]$ $P[1,0]$, видети слику 10.12. Другим речима, сва сабирања и одузимања индекса врше се по модулу n (опсег вредности за индексе i, j је $0,1,\ldots,n-1$). Алгоритам који приказујемо је симетричнији и елегантнији уз ову претпоставку; његово време извршавања једнако је (до на константни фактор) времену извршавања на обичној мрежи (оној која није пресавијањем повезана као торус).

Проблем. Дате су две $n\times n$ матрице A и B, тако да су њихови елементи $A[i,j]$ и $B[i,j]$ у процесору $P[i,j]$. Израчунати $C = AB$, тако да елеменат $C[i,j]$ буде у процесору $P[i,j]$.

Користићемо обичан алгоритам за множење матрица. Проблем је преместити податке тако да се прави бројеви нађу на правом месту у правом тренутку. Посматрајмо елеменат $C[0,0] = \sum_{k=0}^{n-1} A[0,k]\cdot B[k,0]$, који је једнак производу прве врсте матрице A и прве колоне матрице B; редни бројеви врста и колона су при оваквом означавању за један већи од њихових индекса. Волели бисмо да број $C[0,0]$ буде израчунат у процесору $P[0,0]$. Ово се може постићи цикличким померањем прве врсте A улево, и истовременим цикличким померањем прве колоне B увис, корак по корак. У првом кораку



Слика 10.12: Дводимензионална пресавијена мрежа.

$P[0,0]$ садржи $A[0,0]$ и $B[0,0]$ и израчунава њихов производ; у другом кораку $P[0,0]$ добија $A[0,1]$ (од десног суседа) и $B[1,0]$ (од суседа испод себе) и њихов производ додаје парцијалној суми, итд. Вредност $C[0,0]$ се тако израчунава после n корака.

Проблем је обезбедити да сви процесори обављају исти овакав посао, а сви морају да деле податке међусобно. Потребно је да податке преуредимо тако да не само $P[0,0]$, него и сви остали процесори добију све потребне податке. Идеја је да се подаци у матрицама испремештају на такав начин, да сваки процесор у сваком кораку добије два броја чији производ треба да израчуна. Битан је дакле почетни распоред података. Проблем решава почетни распоред такав да процесор $P[i,j]$ има елементе $A[i,i+j]$ и $B[i+j,j]$, односно да други индекс елемента A буде једнак првом индексу елемента B (при чему се, као што је речено, индекси рачунају по модулу n). Пошто се формира овакав распоред, сваки корак се састоји од истовременог цикличког померања врста A и колона B, чиме $P[i,j]$ добија елементе $A[i,i+j+k]$ и $B[i+j+k,j]$, $0 \leq k \leq n-1$, управо оне елементе који су му потребни. До почетног распореда може се доћи цикличким померањем врсте A са индексом i за i места улево, а колоне B са индексом i за i места навише, $i = 0,1,\ldots,n-1$. Алгоритам је приказан на слици 10.13. На слици 10.1 приказано је формирање почетног распореда елемената матрица за $n = 4$. Лева страна показује почетно стање података, а десна њихов размештај после извршавања почетних цикличких померања.

Сложеност. Почетна цикличка померања врста A трају $n/2$ паралелних корака (кад број померања постане већи од $n/2$, померања се изводе у супротном смеру — удесно уместо улево); исто важи и за почетна померања колона B. У наредних n корака изводе се у сваком процесору израчунавања и померања. Ти кораци могу се извршити паралелно. Укупно време извршавања је $O(n)$. Ефикасност алгоритма је $O(1)$, ако упоређујемо паралелни алгоритам са обичним секвенцијалним множењем матрица сложености $O(n^3)$. Ефикасност је асимптотски мања, ако паралелни алгоритам упоређујемо са нпр. Штрасеновим алгоритмом.

Алгоритам Množenje_matrica(A, B);
Улаз: A и B ($n \times n$ матрице).
Излаз: C (производ AB).
begin
for $i := 0$ **to** $n-1$ **do in parallel**
циклички помери улево врсту i матрице A за i места;
{односно, изврши $A[i,j] := A[i,(j+1) \bmod n]$ i пута}
for $j := 0$ **to** $n-1$ **do in parallel**
циклички помери навише колону j матрице B за j места;
{односно, изврши $B[i,j] := B[(i+1) \bmod n, j]$ j пута}
{подаци су сада на жељеним почетним позицијама}
for све парове i, j **do in parallel**
$C[i,j] := A[i,j] \cdot B[i,j]$;
for $k := 1$ **to** $n-1$ **do**
for све парове i, j **do in parallel**
$A[i,j] := A[i,(j+1) \bmod n]$;
$B[i,j] := B[(i+1) \bmod n, j]$;
$C[i,j] := C[i,j] + A[i,j] \cdot B[i,j]$
end

Слика 10.13: Алгоритам за паралелно множење матрица на мрежи.

a_{00} b_{00}	a_{01} b_{01}	a_{02} b_{02}	a_{03} b_{03}
a_{10} b_{10}	a_{11} b_{11}	a_{12} b_{12}	a_{13} b_{13}
a_{20} b_{20}	a_{21} b_{21}	a_{22} b_{22}	a_{23} b_{23}
a_{30} b_{30}	a_{31} b_{31}	a_{32} b_{32}	a_{33} b_{33}

a_{01} b_{10}	a_{02} b_{21}	a_{03} b_{32}	a_{00} b_{03}
a_{12} b_{20}	a_{13} b_{31}	a_{10} b_{02}	a_{11} b_{13}
a_{23} b_{30}	a_{20} b_{01}	a_{21} b_{12}	a_{22} b_{23}
a_{30} b_{00}	a_{31} b_{11}	a_{32} b_{22}	a_{33} b_{33}

Табела 10.1: Почетно размештање елемената матрица — припрема за паралелно множење.

10.5 Систолички алгоритми

Систоличка архитектура личи на покретну траку у фабрици. Процесори су обично распоређени на врло правилан начин (најчешће у облику једнодимензионалног или дводимензионалног поља), а подаци се кроз њих ритмички померају. Сваки процесор извршава једноставне операције са подацима које је добио у претходном кораку, а своје резултате дотура следећој "станици". Сваки процесор може да садржи малу локалну меморију. Улази се најчешће "утискују" у систем један по један, уместо да се сви одједном упишу у неке меморијске локације. Предност систоличке архитектуре је ефикасност, и у хардверском погледу (процесори су специјализовани и једноставни) и у погледу брзине (минимизиран је број приступа меморији). Као и на покретној траци, кључно је да се избегне потреба за довлачењем алата

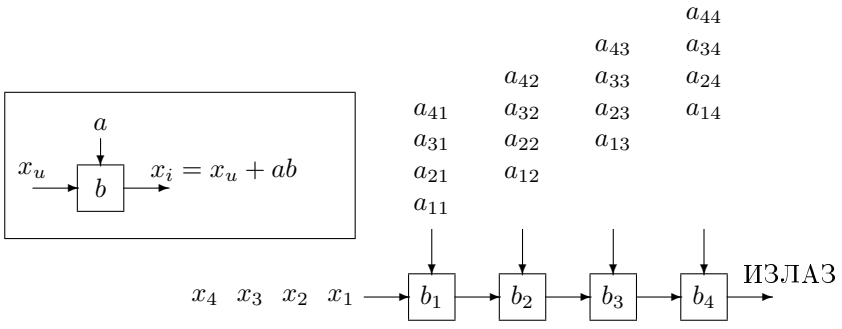
и материјала за време рада. Све што је потребно за извршавање операције, долази покретном траком. Озбиљан недостатак оваквих рачунара је недовољна флексибилност. Систоличке архитектуре су ефикасне само за одређене алгоритме. Размотрићемо два примера систоличких алгоритама.

10.5.1 Множење матрице и вектора

Започињемо са једноставним алгоритмом, који ћемо затим користити за развој компликованијег алгоритма.

Проблем. Израчунати производ $x = Ab$ матрице A димензије $m \times n$ и вектора b дужине n.

Систем се састоји од n процесора, тако да процесор P_i додаје парцијалној суми члан у коме је чинилац b_i, i-та компонента вектора b. Кретање података и операције које извршава сваки од процесора приказани су за $n = m = 4$ на слици 10.14. Претпостављамо да се вектор b налази у одговарајућим процесорима (или је у њих убачен корак по корак). Резултати се акумулирају током кретања слева удесно кроз процесоре. Излазне промењиве x_i на почетку имају вредност 0. У првом кораку се x_1 $(= 0)$ заједно са a_{11} убацује у P_1, а сви остали улази напредују један корак на путу ка процесорима. Процесор P_1 израчунава $x_1 + a_{11} \cdot b_1$, и резултат прослеђује удесно. У другом кораку P_2 прима слева $x_1 = a_{11} \cdot b_1$ и одозго a_{12}, па израчунава $x_1 + a_{12} \cdot b_2$, резултат прослеђује удесно, итд. У i-том кораку израчунавања x_1 процесор P_i прима слева парцијални резултат $x_1 = \sum_{k=1}^{i-1} a_{1k} b_k$, одозго одговарајући елеменат матрице a_{1i}, а одговарајућу координату b_i било из локалне меморије (као на слици), било одоздо. Процесор P_i израчунава вредност израза $x_1 + a_{1i} b_i$ и предаје је даље, удесно. У тренутку напуштања низа процесора, после n корака, x_1 очигледно садржи жељену вредност. Израчунавање x_2 прати " у стопу" израчунавање x_1 и завршава се у $(n+1)$-ом кораку. Другим речима, израчунавања x_1 и x_2 су временски скоро потпуно преклопљена. Уопште, израчунати елеменат x_j појављује се на излазу после $n+j-1$ корака, $j = 1, 2, \ldots, m$, а комплетан производ израчунава се после $m + n - 1$ корака.



Слика 10.14: Множење вектора матрицом.

Основни проблем при конструкцији систоличких алгоритама је организација кретања података. Сваки податак мора се наћи на правом месту у правом тренутку. У овом примеру то је постигнуто увођењем кашњења, тако да

почетак i-те колоне матрице A стиже у процесор P_i у кораку i. Овај пример је једноставан, јер се сваки елеменат матрице A користи само једном. Кад се иста вредност користи више пута, много је компликованије организовати њено кретање, што ћемо видети у следећем примеру.

10.5.2 Израчунавање конволуције

Нека су $x = (x_1, x_2, \ldots, x_n)$ и $w = (w_1, w_2, \ldots, w_k)$ два реална вектора, при чему је $k < n$. **Конволуција** вектора x и w је вектор $y = (y_1, y_2, \ldots, y_{n+1-k})$, са координатама

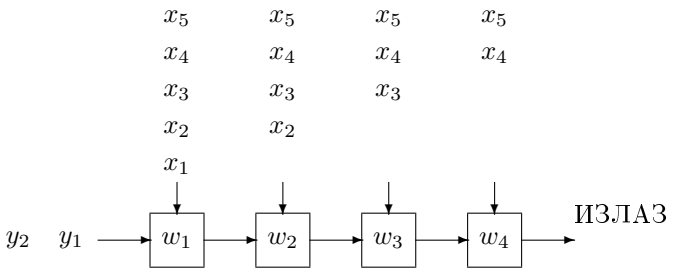
$$y_i = w_1 x_i + w_2 x_{i+1} + \cdots w_k x_{i+k-1}, \quad i = 1, 2, \ldots, n+1-k.$$

Проблем. Израчунати конволуцију y вектора x и w.

Проблем израчунавања конволуције може се свести на проблем израчунавања производа матрице и вектора на следећи начин:

$$\begin{pmatrix} x_1 & x_2 & x_3 & \cdots & x_k \\ x_2 & x_3 & x_4 & \cdots & x_{k+1} \\ x_3 & x_4 & x_5 & \cdots & x_{k+2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{n+1-k} & x_{n+2-k} & x_{n+3-k} & \cdots & x_n \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \cdots \\ w_k \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \cdots \\ y_{n+1-k} \end{pmatrix}$$

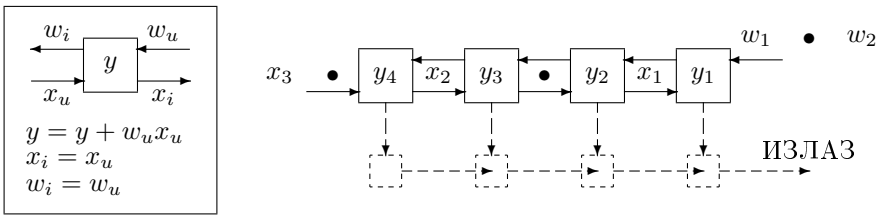
Систолички алгоритам који решава овај проблем може се добити као специјални случај општијег алгоритма за израчунавање производа матрице и вектора (видети претходни проблем, слика 10.14). Ово је приказано на слици 10.15. Приметимо да се x_i истовремено користи дуж целог низа процесора (сем првих $k-1$ вредности x_i, које се користе само у почетном делу низа). Због тога је потребна линија за простирање. Приказаћемо сада решење проблема конволуције без простирања.



Слика 10.15: Конволуција са простирањем.

Процесори на слици 10.15 примају два улаза, а дају само један излаз. Употребићемо сада процесоре који примају улазе из два смера, и шаљу излаз у два смера. Идеја је померати вектор x слева удесно, а вектор w здесна улево. Резултат y акумулира се у процесорима. Кретање података треба тако подесити да се одговарајуће координате w и x сретну тамо где треба да буду помножене. Проблем је у томе што, кад се два вектора крећу један према другом, онда је брзина једног вектора у односу на други двоструко већа. Последица ове чињенице је да би један елеменат

вектора x пропустио половину елемената вектора w, и обрнуто. Решење је померати векторе *двоструко мањом брзином*. Улаз слева је дакле "x_1, ништа, x_2, ништа, ...", а здесна исто то за вектор w. Решење је приказано на слици 10.16 (црне тачке одговарају одсутним подацима).



Слика 10.16: Конволуција помоћу двосмерног низа.

Препуштамо читаоцу да се увери да сваки процесор P_i на крају израчунава вредност y_i. Кад w_k напусти P_i, израчуната је коначна вредност y_i, па се подаци могу "изручити" из низа процесора путем назначеним испод низа процесора на слици 10.16. Недостатак кретања података двоструко мањом брзином је у томе што израчунавање траје двоструко дуже.

10.6 Резиме

Пошто је паралелне алгоритме компликованије конструисати од секвенцијалних, корисно је што више употребљавати готове блокове. Један од таквих моћних блокова је алгоритам за паралелно израчунавање префикса, који је чак предлаган за примитивну машинску инструкцију. Тешко је у овом тренутку проценити које ће од разматраних паралелних архитектура (односно да ли ће нека од њих) доминирати у будућности. Због тога је важно идентификовати технике пројектовања алгоритама заједничке за више модела. Анализирали смо четири такве технике: *удвостручавање* (одређивање рангова елемената листе и друге операције са повезаним листама), *паралелну варијанту разлагања* (сабирање, паралелно израчунавање префикса, сортирање), *преклапање* (код систоличких алгоритама), и *технику Ојлеровог обиласка* (која је корисна код мноштва алгоритама за рад са стаблима, односно графовима).

Литература

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, MIT Press, 2002.

[2] М. Живковић, Алгоритми, Математички факултет, 2000.

[3] E. Horowitz, S. Sahni, S. Rajasekaran, Computer Algorithms, Computer Science Press, 1997.

[4] I. Parberry, W. Gasarch, Problems on Algorithms, Second Edition, 2002.

[5] W. Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees, Communications of the ACM, vol. 33, issue 6, pp. 668–676, 1990.

[6] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

[7] T. Rolfe, P. Purdom, An Alternative Problem for Backtracking and Bounding: The SIGCSE Bulletin 83, Volume 36, Number 4, 2004.