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

9 января 2023 г.

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

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 $\log_2(n!) = \Omega(n \log n)$.

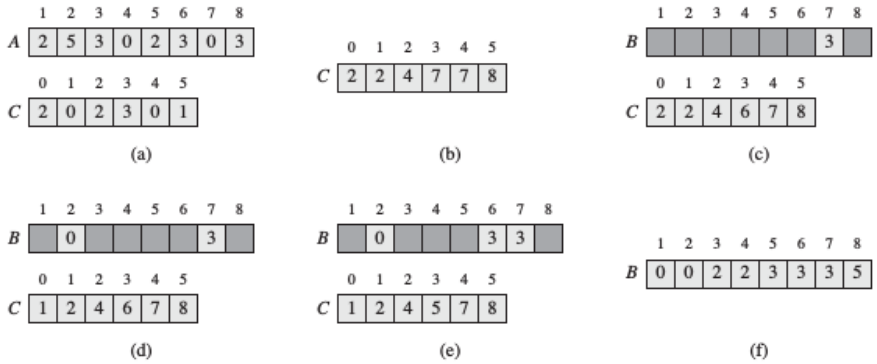
Сортирање је ипак могуће извршавати брже коришћењем специјалних особина бројева, или изводећи алгебарске манипулације са њима. Ово не противречи доказаној доњој граници, јер овакви алгоритми сортирања не користе упоређивања већ рецимо чињеницу да вредности бројева могу да се ефикасно користе као адресе.

2.1 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 постиже да уколико постоји још неки елемент са истом вредношћу



Слика 2.1: 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 2.2: Сортирање пребројавањем.

$A[j]$, један од таквих елемената иде на позицију непосредно пре $A[j]$. На слици 2.1 и у наредној табели приказан је пример сортирања пребројавањем.

i	$A[i]$	$C[A[i]]$	$B[C[A[i]]]$	$C[A[i]]$
8	3	7	3	6
7	0	2	0	1
6	3	6	3	5
5	2	4	2	3
4	0	1	0	0
3	3	5	3	4
2	5	8	5	7
1	2	4	2	3

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.

2.2 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 2.3.

Сложеност. Потребно је n корака за копирање елемената у глобалну листу GL и d корака за иницијализацију листа $Q[i]$. У главној петљи

```
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 2.3: Сортирање директним вишеструким разврставањем.

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 $O(2(n+n)) = O(n)$.

2.3 Сортирање просечне линеарне сложености

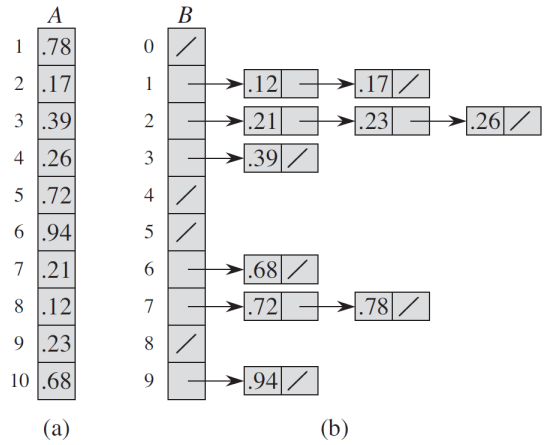
Размотрићемо сада другу варијанту алгоритма. Приказаћемо алгоритам сортирања разврставањем, при чему се претпоставља да елементи улазног низа $A[1..n]$ имају равномерну расподелу на полуотвореном интервалу $[0, 1)$. Прецизније, претпоставка је да је сваки елемент улазног низа генерисан независно од осталих елемената насумичним одабиром елемента из интервала $[0, 1)$.

Сортирање разврставањем разбија интервал $[0, 1)$ на тачно n једнаких подинтервала ("преграде"), 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 преграду. 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 преграде, 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 (преграде); претпоставља се да постоји механизам за одржавање таквих листи. Алгоритам је приказан на слици 2.4. Пример извршавања овог алгоритма приказан је на слици 2.5.

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 2.4: Сортирање разврставањем.



Слика 2.5: 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.

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 преграду као и $A[j]$, или је смештен у преграду са мањим индексом.

Ако се та два елемента сместе у исту преграду, онда их последња петља у алгоритму премешта у исправан редослед. Ако се та два елемента сместе у различите преграде, онда их последња наредба у алгоритму смешта у исправан редослед. Дакле, предложени алгоритам ради коректно.

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

$$E[n_i^2] = 2 - 1/n, \tag{2.1}$$

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:

$$\begin{aligned}
E[n_i^2] &= E\Big[\Big(\sum_{j=1}^{n} X_{ij}\Big)^2\Big] \\
&= E\Big[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}\cdot X_{ik}\Big] \\
&= E\Big[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\neq j}} X_{ij}\cdot X_{ik}\Big] \\
&= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\neq j}} E[X_{ij}\cdot X_{ik}] \qquad (2.2)
\end{aligned}$$

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:

$$E[X_{ij}^2] = 1^2\cdot\frac{1}{n} + 0^2\cdot\big(1-\frac{1}{n}\big) = \frac{1}{n} \qquad (2.3)$$

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

$$E[X_{ij}\cdot X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n}\cdot\frac{1}{n} = \frac{1}{n^2} \qquad (2.4)$$

Заменом ове две вредности у једначину 2.2, добијамо:

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

чиме је доказана једнакост 2.1.

Коришћењем добијеног очекивања из једначине 2.1, 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 2.1 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 3

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.

3.1 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 3.2.

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 3.3 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.

3.2 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.

3.3 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.

Пример улаза за проблем приказан је у наредној табели.

подскуп ēлеменат	1	2	3	4	5	6	7	8
1		x	x		x	x		x
2			x	x		x	x	
3	x			x		x	x	

Бојење које задовољава тај услов зваћемо *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 4

Суфиксни низ и суфиксно стабло

4.1 Увод

Суфиксни низ и суфиксно стабло дате ниске су структуре података помоћу којих се ефикасно могу решавати проблеми као што је тражење речи у тексту, тражење понављања или палиндрома у тексту, тражење заједничке подниске за два или више текстова и слично.

Нека је задата ниска s дужине n. Нека је $s[i..j]$ део те ниске са индексима од i до j, $1 \leq i \leq j \leq n$. Специјално, нека $S_i = s[i..n]$ означцава суфикс ниске s који почиње од индекса i. Суфиксни низ SA ниске s је низ индекса лексикографски сортираних суфикса: $S_{SA[1]} < S_{SA[2]} < \cdots < S_{SA[n]}$. Другим речима, ако је $SA[i] = j$, онда суфикс S_j има ранг $i = rang[j]$ међу нискама $S_1, S_2, \ldots, S_n$. Низови SA и $rang$ су међусобно инверзне пермутације скупа $\{1, 2, \ldots, n\}$: $SA[rang[j]] = rang[SA[j]] = j$, $1 \leq j \leq n$.

Пример 4.1. Сортирани низ суфикса ниске $s =$ mississippi и њен суфиксни низ приказани су у табели 4.1. Ниска $S_8 =$ ippi је суфикс $s[8..11]$.

- Лексикографски редослед суфикса је $S_{11} < S_8 < \cdots < S_3$.
- Који је по реду суфикс S_5? Трећи (његов ранг је 3), јер је $rang(5) = 3$.
- Који суфикс је седми по реду? То је суфикс $S_9 =$ ppi, јер је $SA[7] = 9$.

Најдужи заједнички префикс или ЛЦП (longest common preffix)за две ниске s_1 и s_2 је дужина $LCP(s_1, s_2)$ најдуже ниске која је префикс и s_1 и s_2. За фиксирану ниску s нека је $LCP(i, j) = LCP(S_i, S_j)$. Нисци s може се поред њеног суфиксног низа SA придружити и низ LCP, чији је i-ти члан једнак $LCP[i] = LCP(S_{SA[i-1]}, S_{SA[i]})$, $i = 2, 3, \ldots, n$. Другим речима, $LCP[i]$ је дужина најдужег заједничког префикса i-тог и $(i+1)$-ог суфикса ниске у сортираном редоследу.

Пример 4.2. Низ LCP ниске $s =$ mississippi приказан је у табели 4.1 заједно са суфиксним низом. За ову ниску је $LCP(4) = 4$, јер је $SA[3] = 5$, $S_5 = issippi$, $SA[4] = 2$, $S_2 = ississippi$, и

$$LCP[4] = LCP(issippi, ississippi) = |issi| = 4.$$

i	$SA[i]$	$rang[i]$	$S_{SA[i]}$	$LCP[i]$	Суфиксно стабло s
1	11	5	i		i 11
2	8	4	ippi	1	ppi 8
3	5	11	issippi	1	ssi ippi 5
4	2	9	ississippi	4	ssippi 2
5	1	3	mississippi	0	mississippi 1
6	10	10	pi	0	p i 10
7	9	8	ppi	1	pi 9
8	7	2	sippi	0	s i ppi 7
9	4	7	sissippi	2	ssippi 4
10	6	6	ssippi	1	si ppi 6
11	3	1	ssissippi	3	ssippi 3

Table 4.1: Суфиксни низ, низ LCP и суфиксно стабло ниске миссиссиппи.

Суфиксно стабло је друга структура података за представљање интерне структуре ниске. Суфиксно стабло ниске s дужине n је коренско стабло за које важи:

- стабло има тачно n листова који су нумерисани бројевима $1, 2, \ldots, n$;
- сваки унутрашњи чвор има бар два сина;
- свака грана је означена непразном подниском ниске s;
- никоје два гране из истог чвора немају ознаке које почињу истим карактером;
- конкатенација свих ознака на путу од корена до листа са ознаком i једнака је суфиксу $S_i = s[i..n]$.

Пример 4.3. Суфиксно стабло ниске $s =$ mississippi приказано је у табели 4.1 заједно са њеним суфиксним низом.

Пошто је $|s| = 11$, суфиксно стабло има 11 листова. Поред тога, ово стабло има корен и 7 унутрашњих чворова. Конкатенација ознака грана на путу од корена до листа 7 је $S_7 =$ i $-$ pi $-$ ppi.

Из дефиниције не следи постојање суфиксног стабла за произвољну ниску s. Проблем се појављује када је неки суфикс S_i једнак префиксу неког другог суфикса S_j, јер се тада пут који одговара суфиксу S_i завршава унутар пута који одговара суфиксу S_j, тј. не завршава се у листу стабла. На пример, не постоји суфиксно стабло за ниску cxabxa, јер се пут који одговара суфиксу ша, као префикс суфикса шабша, не завршава у листу. Уобичајени начин да решавања овог проблема реши је да се на крај ниске дода знак који се не појављује унутар ње (обично је то знак \$; претпоставља се да знак \$ лексикографски претходи свим осталим знацима).

Лема 4.1. Укупан број грана у суфиксном стаблу ниске дужине n је највише $2n - 1$.

Доказ. Можемо да замислимо да суфиксно стабло настаје од корена и само једне гране која води до јединог листа додавањем нових суфикса. Пошто додавање сваког унутрашњег чвора повећава број листова бар за 1, а број листова у коначном стаблу је n, број унутрашњих чворова је највише $n-1$. Број грана ка листовима једнак је n, а број грана ка унутрашњим чворовима је највише $n-1$. Према томе, укупан број грана у суфиксном стаблу је највише $2n-1$.

Ако би се у суфиксном стаблу уз сваку грану чувала њена комплетна ознака, онда би просторна сложеност стабла за ниску дужине n била у најгорем случају $O(n^2)$. Међутим, произвољна подниска $s[i..j]$ познате ниске s одређена је са два индекса i и j. Пошто је на основу доказане леме број грана у суфиксном стаблу ниске дужине n највише $O(n)$, заменом ознака на гранама са по два броја постиже се да је простор који заузима суфиксно стабло $O(n)$.

4.2 Алгоритам линеарне сложености за конструкцију суфиксног низа

Без смањења општости може се претпоставити се да су знакови ниске $s = s[1..n]$ кодирани природним бројевима из скупа $\{1, 2, \ldots, n\}$. Упознаћемо се са алгоритмом линеарне сложености за конструкцију суфиксног низа ниске чији су аутори Каркаинен и Сандерс [8]. Тај алгоритам је изложен у последњем издању "библије" [9]. Заснован је на специфичном разлагању (divide-and-conquer):

1. Сортирају се суфикси S_i ниске s, $i \mod 3 \neq 0$ ($i = 1, 2, 4, 5, 7, 8, 10, 11, \ldots$; у даљем тексту: А-суфикси) свођењем на рекурзивну конструкцију суфиксног низа посебно формиране ниске дужине око $2n/3$.

2. На основу тога се сортирају преостали суфикси S_i, $i \mod 3 = 0$ ($i = 3, 6, 9, \ldots$; у даљем тексту: Б-суфикси).

3. Обједињавањем сортираних низова А-суфикса и Б-суфикса одређује се суфиксни низ ниске s.

Прелазимо на детаљнији опис три основна корака алгоритма, илуструјући поступак обрадом ниске $s =$ mississippi.

1. Полазећи од ниске s, формира се ниска P над азбуком од мета знакова — трословних блокова полазне азбуке; при томе суфикси ниске P имају исти лексикографски редослед (одређен суфиксним низом):

   А Формирати ниске P_1 и P_2 од мета знакова (тројке одвајамо заградама):

   - $P_1 = (s[1..3])(s[4..6])(s[7..9]) \cdots (s[n'..n'+2])$, где је n' највећи број са остатком 1 по модулу 3, $n' \leq n$. При томе је ниска s продужена специјалним знаковима $\varnothing$ са кодом 0. За $s =$ mississippi добија се $P_1 = (mis)(sis)(sip)(pi\varnothing)$.
   - $P_2 = (s[2..4])(s[5..8])(s[8..10]) \cdots (s[n''..n''+2])$, где је n'' највећи број са остатком 2 по модулу 3, $n'' \leq n$. За ниску s је $P_2 = (iss)(iss)(ipp)(i\varnothing\varnothing)$.

Ако је n дељиво са 3, на крај P_1 додаје се мета знак ($\varnothing\varnothing\varnothing$). Тиме се постиже да се P_1 увек завршава мета знаком који садржи $\varnothing$ (ово не мора да важи за ниску P_2).

B Нека је P конкатенација ниски P_1 и P_2. чињеница да се P_1 завршава мета знаком који садржи $\varnothing$ обезбеђује да су два суфикса P, од којих један почиње унутар P_1, а други унутар P_2, увек различита. Од индекса i' мета знака $P[i'] = s[i..i+2]$, одговарајући индекс $i = g(i')$, $i \equiv 1, 2 \pmod 3$, унутар ниске s добија се на следећи начин:

$$i = g(i') \begin{cases} 3i' - 2, & i' \leq |P_1| \\ 3i' - 1 - 3|P_1|, & i' > |P_1| \end{cases} \tag{4.1}$$

Обрнуто:

$$i' = g^{-1}(i) = \begin{cases} (i+2)/3, & i \equiv 1 \pmod 3 \\ (i+1)/3 + |P_1|, & i \equiv 2 \pmod 3 \end{cases} \tag{4.2}$$

У табели 4.2 приказана је ниска P. При томе је на пример

- $P[3] = \text{sip} = s[7..9]$, јер је $7 = g(3) = 3 \cdot 3 - 2$ ($3 \leq |P_1| = 4$), односно $3 = (7+2)/3$ ($7 \equiv 1 \pmod 3$);
- $P[6] = \text{iss} = s[5..7]$, јер је $5 = g(6) = 3 \cdot 6 - 1 - 3|P_1|$ ($5 > |P_1| = 4$), односно $6 = (5+1)/3 + |P_1|$ ($5 \equiv 2 \pmod 3$).

Запажа се да је лексикографски редослед суфикса ниске P исти као редослед одговарајућих A-суфикса ниске s.

C Сортирати троструким разврставањем и рангирати јединствене мета знаке ниске P (дакле, занемарујући дупликате), рачунајући рангове од 1. Сложеност овог сортирања је $O(n)$. У нашем примеру P има 7 јединствених мета знакова. Њихов сортирани редослед је (и$\varnothing\varnothing$), (ipp), (iss), (iss), (mis), (pi$\varnothing$), (sip), (sis).Рангови ових мета знакова су редом $1, 2, \ldots, 7$. Мета знак (исс) се у нисци P једини појављује два пута.

D Као у табели 4.2, формира се нова ниску P' кодирањем мета знакова њиховим управо одређеним ранговима. Ако P садржи k јединствених мета знакова, онда је сваки "знак" у P' неки природни број од 1 до k. Суфиксни низови ниски P и P' су идентични.

E Одредити суфиксни низ $SA_{P'}$ ниске P'. Ако су сви знаци у P' јединствени, онда се суфиксни низ добија директно, јер редослед појединих знакова одређује суфиксни низ. У противном, одредити рекурзивно суфиксни низ P', третирајући рангове у P' као знакове ниске. У табели 4.2 је приказан суфиксни низ $SA_{P'}$ добијен у нашем примеру. Пошто је број мета знакова у P', а тиме и дужина P', отприлике $2n/3$, рекурзивни проблем је мањи од полазног.

F На основу $SA_{P'}$ и позиција A-суфикса у нисци s, одредити списак рангова (позиција) сортираних A-суфикса у s. Прецизније, за $i = 1, 2, \ldots, |P|$ израчунати на основу (4.1) индекс $j = g(SA_{P'}[i])$ и ставити $r_j \leftarrow i$. У табели 4.3 приказани су рангови r_j сортираних A-суфикса $s[i..i+2]$ у нашем примеру. Другим речима, резултат сортирања A-суфикса је

$$S_{11} < S_8 < S_5 < S_2 < S_1 < S_{10} < S_7 < S_4$$

2. Број Б-суфикса је око 1/3 броја свих суфикса. Користећи сортирани редослед А-суфикса, сортирати Б-суфиксе примењујући следећи поступак.

   G Проширивши ниску s са два знака $\varnothing\varnothing$ (после чега је дужина ниске $n+2$), посматрајмо суфиксе $s[i..n+2]$ за $i = 1, 2, \ldots, n+2$. Доделити сваком суфиксу $s[i..n+2]$ његов ранг r_i.

   - За два специјална знака $\varnothing\varnothing$ ставити $r_{n+1} = r_{n+2} = 0$.
   - За А-суфиксе у нисци s рангови су додељени у кораку Ф.
   - Рангови Б-суфикса тренутно нису дефинисани, па за њих ставити $r_i = \square$.

   У табели 4.3 приказани су рангови за ниску $s =$ mississippi и $n = 11$.

   H Сортирати Б-суфиксе двоструким разврставањем (сложености $O(n)$), замењујући их (јединственим!) паровима $(s[i], r_{i+1})$. У нашем примеру добија се

   $$S_9 < S_6 < S_3,$$

   јер је $(p, 6) < (s, 7) < (s, 8)$.

3. Објединити сортиране низове А-суфикса и Б-суфикса. Нека су S_i и S_j нека два суфикса која треба упоредити у току обједињавања, при чему је S_i А-суфикс, а S_j Б-суфикс. У зависности од тога да ли је $i \bmod 3$ једнако 1 или 2, резултат упоређивања суфикса S_i и S_j исти је као резултат упоређивања

   - парова $(s[i], r_{i+1})$ и $(s[j], r_{j+1})$, односно
   - тројки $(s[i], s[i+1], r_{i+2})$ и $(s[j], s[j+1], r_{j+2})$.

   У нашем примеру приликом обједињавања се врши следећи низ упоређивања А- и Б-суфикса:

   - $S_{11} < S_9$ јер је $(i, \varnothing, 0) < (p, p, 1)$; излаз S_{11};
   - $S_8 < S_9$ јер је $(i, p, 6) < (p, p, 1)$; излаз S_8;
   - $S_5 < S_9$ јер је $(i, s, 7) < (p, p, 1)$; излаз S_5;
   - $S_2 < S_9$ јер је $(i, s, 8) < (p, p, 1)$; излаз S_2;
   - $S_1 < S_9$ јер је $(m, 4) < (p, 6)$; излаз S_1;
   - $S_{10} < S_9$ јер је $(p, 1) < (p, 6)$; излаз S_{10};
   - $S_7 > S_9$ јер је $(s, 2) < (p, 6)$; излаз S_9;
   - $S_7 < S_6$ јер је $(s, 2) < (s, 7)$; излаз S_7;
   - $S_4 < S_6$ јер је $(s, 3) < (s, 7)$; излаз S_4;
   - преостала два суфикса S_6, S_3 копирају се у излазни низ.

   Тиме је одређен сортирани редослед свих суфикса ниске s, односно њен суфиксни низ, видети табелу 4.1. Пошто је сложеност упоређивања $O(1)$, сложеност обједињавања је $O(n)$.

Индекс i' у P, P'	1	2	3	4	5	6	7	8
Индекс $i = g(i')$ у s	1	4	7	10	2	5	8	11
$P[i'] = s[i..i+2]$	(mis)	(sis)	(sip)	(pi∅)	(iss)	(iss)	(ipp)	(i∅∅)
$P'[i']$	4	7	6	5	3	3	2	1
$SA_{P'}[i'] = SA_P[i']$	8	7	6	5	1	4	3	2
Индекс i за који је ранг$r_i = i'$	11	8	5	2	1	10	7	4

Table 4.2: Поступак сортирања A-суфикса ниске s = mississippi у првој фази одређивања суфиксног низа.

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$s[i]$	m	i	s	s	i	s	s	i	p	p	i	∅	∅
$r_i = i'$	5	4	□	8	3	□	7	2	□	6	1	□	0

Table 4.3: Рангови r_1 до r_{n+3} за ниску s = mississippi, $n = 11$.

Сложеност алгоритма задовољава диференцну једначину $T(n) = O(n) + T(\lfloor 2n/3 \rfloor)$, чије је решење на основу мастер теореме $T(n) = O(n)$.

Ако претпоставимо да је време извршавања алгоритма приближно $T(n) = cn$, заменом у диференцној једначини

- $T(n)$ са cn,
- $O(n)$ са an, и
- $T(2n/3)$ са $2cn/3$,

долази се до једначине $cn = an + 2cn/3$, из које следи да је $a = c/3$. Закључује се да рекурзивно сортирање A-суфикса траје приближно око 2/3 времена потребног за одређивање суфиксног низа комплетне ниске.

4.3 Одређивање низа LCP на основу суфиксног низа

Низ LCP дате ниске s може се одредити алгоритмом линеарне сложености који је предложило неколико аутора 2001. године [11]. Подсетимо се да је $LCP[i]$ дужина најдужег заједничког префикса $(i-1)$-ог и i-тог лексикографски најмањег суфикса $S_{SA[i-1]}$ и $S_{SA[i]}$ ниске s. По дефиницији је $LCP[1] = 0$. Приликом израчунавања низа LCP користи се низ $rang$, који садржи инверзну пермутацију пермутације SA: ако је $SA[i] = j$, онда је $rang[j] = i$. Другим речима, $rang[SA[i]] = i$ за $i = 1, 2, \ldots, n$. За суфикс S_i број $rang[i]$ је позиција тог суфикса у међу лексикографски сортираним суфиксима.

Пример 4.4. Видели смо да је суфиксни низ ниске mississippi пермутација

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 11 & 8 & 5 & 2 & 1 & 10 & 9 & 7 & 4 & 6 & 3 \end{pmatrix}$$

Низ $rang$ је инверзна пермутација ове пермутације.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 5 & 4 & 11 & 9 & 3 & 10 & 8 & 2 & 7 & 6 & 1 \end{pmatrix}$$

На пример, суфикс сиссиппи је S_4. Његова позиција у сортираном редоследу суфикса је $rang[4] = 9$.

У току рачунања низа LCP потребно је одредити где се у лексикографском редоследу налази суфикс који се од текућег суфикса добија брисањем првог знака. За ово је користан низ $rang$. Посматрајмо i-ти најмањи суфикс, $S_{SA[i]}$. Брисањем његовог првог знака добија се суфикс $S_{SA[i]+1}$, као суфикс који почиње од индекса $SA[i] + 1$ ниске s. Позиција тог суфикса у сортираном редоследу је $rang[SA[i] + 1]$.

Пример 4.5. Наставак претходног примера. Потребно је за суфикс $S_4 =$ sissippi одредити где се налази суфикс иссиппи (сиссиппи без првог знака). Суфикс сиссиппи је на позицији $rang[4] = 9$ у суфиксном низу и $SA[9] = 4$. Пошто је $rang[SA[9] + 1] = rang[5] = 3$, наредни суфикс иссиппи је на позицији 3 у сортираном редоследу.

Низ LCP може се одредити алгоритмом чији код је приказан на следећој страни.

```
Odredjivanje LCP(s, SA, n)
// алоцирати простор за низове rang[n] и LCP[n]
for i ← 1 to n do
  rang[SA[i]] ← i      // по дефиницији
LCP[1] ← 0      // по дефиницији
l ← 0      // текући елемент низа
for i ← 0 to n do
  if rang[i] > 1      // тада се одређује LCP[rang[i]]
    j ← SA[rang[i] − 1]      //S_j је лексикографски претходник S_i
    m ← max{i, j}
    while m + l < n and s[i + l] = s[j + l] do
      l ← l + 1      // наредни знак у заједничком префиксу
    LCP[rang[i]] ← l      // LCP(S_j, S_i)
    if l > 0
      l ← l − 1      // уклонити први знак у заједничком префиксу S_i, S_j
return LCP
```

Коректност алгоритма заснована је на наредној леми.

Лема 4.2. Посматрајмо суфиксе S_{i-1} и S_i, чије су позиције у лексикографском редоследу суфикса редом $rang[i-1]$ и $rang[i]$. Ако је $LCP[rang[i-1] = l > 1$, онда за суфикс S_i, који се од суфикса S_{i-1} добија брисањем првог знака, важи $LCP[rang[i]] \geq l - 1$.

Доказ. Суфикс S_{i-1} је на позицији $rang[i-1]$ у сортираном редоследу суфикса. Суфикс који му непосредно претходи је у сортираном редоследу на позицији $rang[i-1]-1$, и то је суфикс $S_{rang[i-1]-1}$. По претпоставци је $LCP(S_{SA[rang[i-1]-1]}, S_{i-1}) = l > 1$. Уклањањем првог знака у овим суфиксима добијају се суфикси $S_{SA[rang[i-1]-1]+1}$ и S_i, за које је најдужи заједнички префикс дужине $l-1$.

- Ако суфикс $S_{SA[rang[i-1]-1]+1}$ непосредно претходи суфиксу S_i у сортираном редоследу, доказ је завршен.

- Претпоставимо зато да суфикс $S_{SA[rang[i-1]-1]+1}$ не претходи непосредно суфиксу S_i у сортираном редоследу. Суфикс $S_{SA[rang[i-1]-1]}$ непосредно претходи суфиксу S_{i-1}, и они имају једнаких првих $l > 1$ знакова, па суфикс $S_{SA[rang[i-1]-1]+1}$ мора да буде у сортираном редоследу пре суфикса S_i. Између њих стоји неколико суфикса. Сваки такав суфикс почиње са истих $l-1$ знакова као $S_{SA[rang[i-1]-1]+1}$ и S_i. Према томе, који год од ових суфикса стоји на позицији $rang[i]-1$ непосредно испред S_i, он има бар $l-1$ првих знакова истих као S_i. Према томе, $LCP[rang[i]] \geq l-1$.

Објашњење кода алгоритма. У првој **for** петљи се израчунава низ $rang$. Први елемент низа LCP је по дефиницији $LCP[1] = 0$. Друга **for** петља израчунава низ LCP пролазећи суфиксе према опадајућим дужинама.

Инваријанта петље је следеће тврђење:

- У линији $LCP[rang[i]] = l$ одређују се елементи низа LCP са индексима редом $rang[1]$, $rang[2]$, $rang[3]$, ..., $rang[n]$.
- Пре те линије суфикс S_i и суфикс S_j који му непосредно претходи имају најдужи заједнички префикс дужине бар l.

Ово тврђење је тачно у првом проласку кроз **for** петљу, јер је тада $l = 0$. Ако се претпостави да се у линији $LCP[rang[i]] = l$ одређује исправна вредност $LCP[rang[i]]$, онда се та индуктивна претпоставка због декрементирања l (ако је позитивно), одржава на основу доказане леме. Најдужи заједнички префикс суфикса S_i и S_j може бити дужи од вредности l на почетку итерације, па се тачна вредост l одређује у **while** петљи. Индекс m користи се да обезбеди да тест $s[i+l] = s[j+l]$ не изађе ван граница ниске. После изласка из **while** петље вредност l једнака је дужини најдужег заједничког префикса суфикса S_i и S_j.

Пример 4.6. На примеру ниске mississippi могу се видети три различита случаја на које се може наићи.

- У нисци mississippiсуфикси $S_5 = issippi$ $S_2 = ississippi$ су узастопни у суфиксном низу (трећи и четврти, $SA[4] = 2$, $SA[3] = 5$). Дужина њиховог најдужег заједничког префикса је $LCP[4] = 4$. Када им се обришу први знаци, добијају се суфикси $S_6 = ssippi$ $S_3 = ssissippi$ који су такође узастопни у суфиксном низу (десети и једанаести, $SA[10] = 6$, $SA[11] = 3$). Овде је $LCP[11] = lcp[4] - 1$.
- Суфикси $S_{10} = pi$ и $S_9 = ppi$ су такође узастопни у суфиксном низу (шести и седми, $SA[6] = 10$, $SA[7] = 9$). Дужина њиховог најдужег заједничког префикса је $LCP[7] = 1$. Када им се обришу први знаци, добијају се суфикси $S_{11} = i$ $S_{10} = pi$ који НИСУ узастопни у суфиксном низу (први и шести, $SA[1] = 11$, $SA[6] = 10$). Због тога је $1 = LCP[2] \geq LCP[7] - 1 = 0$; запажамо да у овом примеру важи строга неједнакост $LCP[2] = 1 > LCP[7] - 1 = 0$
- Суфикси $S_{11} = i$ и $S_8 = ippi$ су такође узастопни у суфиксном низу (први и други, $SA[1] = 11$, $SA[2] = 8$). Дужина њиховог најдужег заједничког префикса је $LCP[2] = 1$. Када се суфиксу $S_{11} = i$ обрише

први знак, добија се празан суфикс, кога нема у нашем сортираном редоследу суфикса. Због тога се вредност $LCP[2] = 1$ не може искористити на описани начин за одређивање неке друге вредности $LCP[.]$.

Низ *rang* контролише редослед одређивања елемената низа LCP.

- Полази се од $LCP[rang[1]] = LCP[5]$; упоређују се суфикси са индексима $SA[rang[1]] = 1$ и $SA[rang[1]-1] = 2$, тј. суфикси миссиссиппи и иссиссиппи. Први знаци ова два суфикса се разликују, па је $LCP[5] = 0$.
- Даље се редом одређују вредности $LCP[rang[i]]$, $i = 2, 3, \ldots, 10$, користећи као почетну вредност индекса за упоређивање $l + 1 = LCP[rang[i] - 1] + 1$.
- На пример, пошто је установљено да је $LCP[rang[2]] = LCP[4] = 4$, прелази се на одређивање $LCP[rang[3]] = LCP[11] \geq 4 - 1 = 3$. Упоређују се, почевши од индекса $l + 1 = 4$ суфикси ссиссиппи и ссиппи са индексима редом $SA[rang[3]] = SA[11] = 3$ и $SA[rang[3] - 1] = SA[10] = 6$; упоређујући (различите) четврте знакове ова два суфикса, закључује се да је $LCP[11] = 3$.

Сложеност алгоритма. Почетна вредност l је 0; l ни у једном тренутку не прелази n. Пошто је број декрементирања l највише $n-1$, број инкрементирања l унутар **while** петље је највише $2n$. Према томе, сложеност алгоритма је $O(n)$.

4.4 Примене суфиксног низа

Многе операције са нискама ефикасно се извода ако се претходно формира њихов суфиксни низ и низ LCP. Приказаћемо два таква примера примене: тражење речи у тексту и тражење најдуже заједничке подниске за две дате ниске. На основу суфиксног низа и низа LCP може се алгоритмом линеарне сложености формирати суфиксно стабло ниске, што затим омогућује сличне примене суфиксног стабла.

4.4.1 Тражење речи у тексту

Потребно је одредити све појаве речи P унутар текста S. Проблем је лако решити ако се одреди суфиксни низ текста S: свака појава P унутар S је почетак неког префикса S. Суфиксни низ текста S одређује сортирани редослед суфикса S, па се прва појава P унутар S може пронаћи бинарном претрагом опсега $[1..|S|]$. Индекси свих осталих појава (ако их има) налазе се на наредним узастопним позицијама у суфиксном низу.

Сложеност алгоритма је $O(n \log n)$, јер је сложеност формирања суфиксног низа $O(n)$, а сложеност сваког од $O(\log n)$ упоређивања је $O(n)$. Упоређивања се могу **извршити ефикасније** ако се користи низ LCP текста S.

Пример 4.7. Пронаћи све појаве речи $P =$ *lednik* унутар текста $s =$ *prestolonaslednikovica*. Суфиксни низ текста s (са сортираним редоследом суфикса) приказан је у следећој табели. Бинарна претрага започиње

упоређивањем $P = lednik$ са $S[SA[\lceil \frac{1+22}{2} \rceil]] = S[SA[12]] = naslednikovica$. Пошто је $naslednikovica > lednik$, наставља се са бинарном претрагом опсега $[1, 11]$. После неколико корака проналази се да је индекс прве појаве P унутар s једнак 12. Пошто наредни суфикс $s[13] = ednikovica$ не започиње са P, унутар s нема других појава P.

i	S_i	i	$SA[i]$	$S_SA[i]$	$LCP[i]$
1	prestolonaslednikovica	1	22	a	1
2	restolonaslednikovica	2	10	aslednikovica	0
3	estolonaslednikovica	3	21	ca	0
4	stolonaslednikovica	4	14	dnikovica	0
5	tolonaslednikovica	5	13	ednikovica	0
6	olonaslednikovica	6	3	estolonaslednikovica	0
7	lonaslednikovica	7	20	ica	1
8	onaslednikovica	8	16	ikovica	0
9	naslednikovica	9	17	kovica	0
10	aslednikovica	10	12	**lednik**ovica	1
11	slednikovica	11	7	lonaslednikovica	0
12	lednikovica	12	9	naslednikovica	1
13	ednikovica	13	15	nikovica	0
14	dnikovica	14	6	olonaslednikovica	1
15	nikovica	15	8	onaslednikovica	1
16	ikovica	16	18	ovica	0
17	kovica	17	1	prestolonaslednikovica	0
18	ovica	18	2	restolonaslednikovica	0
19	vica	19	11	slednikovica	1
20	ica	20	4	stolonaslednikovica	0
21	ca	21	5	tolonaslednikovica	0
22	a	22	19	vica	

4.4.2 Тражење најдуже заједничке подниске за две дате ниске

Потребно је одредити најдужу заједничку подниску две дате ниске s_1 и s_2. Проблем се решава тако што се најпре формира суфиксни низ ниске $s = s_1\$s_2\#$, при чему су знаци \$ и # знаци који се не појављују унутар s_1 или s_2. При томе се сваком суфиксу претходно придружује податак да ли је његов почетак унутар s_1 или s_2. У сортираном низу суфикса ове ниске потребно је пронаћи пар узастопних чланова таквих да

- одговарајући суфикси не припадају истом низу, и
- дужина њиховог најдужег заједничког префикса је **највећа у односу** на све остале овакве парове индекса (оваквих парова префикса са најдужим заједничким префиксом може бити више).

Ако се користи и низ LCP здружене ниске s (сложеност његовог формирања је $O(n)$; овде је $n = |s_1|+|s_2|$), онда је сложеност алгоритма линеарна, $O(n)$.

Пример 4.8. Пронаћи најдужу заједничку подниску ниски

$$s_1 = prestolonaslednikovica$$

и

$$s_2 = kolonizacija.$$

Суфиксни низ SA здружене ниске $s = s_1\$s_2\#$ (са сортираним редоследом суфикса) приказан је у табели на следећој страни. Највећи број у колони $LCP[i]$ је $LCP[26] = 4$. Поред тога, суфикси

$$s[SA[26]] = olonaslednikovica\$kolonizacija\#$$

и

$$s[SA[27]]olonizacija\#$$

са заједничким префиксом *olon* дужине четири потичу из различитих ниски. Према томе, најдужа заједничка подниска ове две ниске је ниска *olon*.

4.4.3 Проналажење најдужег палиндрома у задатој нисци

Нека је задата ниска s и нека је потребно пронаћи најдужи палиндром унутар ње. Решавање овог проблема своди се на тражење најдуже заједничке подниске ниски s и s', где је s' ниска која се од s добија "читањем уназад".

4.4.4 Формирање суфиксног стабла

Ако се зна суфиксни низ SA и ЛЦП низ LCP ниске s дужине n, суфиксно стабло те ниске може се конструисати алгоритмом сложености $O(n)$ на основу следеће идеје: полазећи од парцијалног суфиксног стабла за лексикографски најмањи суфикс, уметати у њега остале суфиксе редом којим се на њих наилази лек-сикографским обиласком суфиксног стабла.

	S_i	i		$SA[i]$	$s[SA[i]]$	$LCP[i]$
1	prestolonaslednikovica$ko...#	1	2	37		0
1	restolonaslednikovica$ko...#	2	2	36	#	0
1	estolonaslednikovica$ko...#	3	1	23	$ko...#	0
1	stolonaslednikovica$ko...#	4	2	35	a#	1
1	tolonaslednikovica$ko...#	5	1	22	a$ko...#	1
1	olonaslednikovica$ko...#	6	2	31	acija#	1
1	lonaslednikovica$ko...#	7	1	10	aslednikovica$ko...#	0
1	onaslednikovica$ko...#	8	1	21	ca$ko...#	1
1	naslednikovica$ko...#	9	2	32	cija#	0
1	aslednikovica$ko...#	10	1	14	dnikovica$ko...#	0
1	slednikovica$ko...#	11	1	13	ednikovica$ko...#	1
1	lednikovica$ko...#	12	1	3	estolonaslednikovica$ko...#	0
1	ednikovica$ko...#	13	1	20	ica$ko...#	1
1	dnikovica$ko...#	14	2	33	ija#	1
1	nikovica$ko...#	15	1	16	ikovica$ko...#	1
1	ikovica$ko...#	16	2	29	izacija#	0
1	kovica$ko...#	17	2	34	ja#	0
1	ovica$ko...#	18	2	24	kolonizacija#	2
1	vica$ko...#	19	1	17	kovica$ko...#	0
1	ica$ko...#	20	1	12	lednikovica$ko...#	1
1	ca$ko...#	21	1	7	lonaslednikovica$ko...#	3
1	a$ko...#	22	2	26	lonizacija#	0
1	$ko...#	23	1	9	naslednikovica$ko...#	1
2	kolonizacija#	24	1	15	nikovica$ko...#	2
2	olonizacija#	25	2	28	nizacija#	0
2	lonizacija#	26	**1**	6	**olon**aslednikovica$ko...#	**4**
2	onizacija#	27	**2**	25	**olon**izacija#	1
2	nizacija#	28	1	8	onaslednikovica$ko...#	2
2	izacija#S	29	2	27	onizacija#	1
2	zacija#	30	1	18	ovica$ko...#	0
2	acija#	31	1	1	prestolonaslednikovica$ko...#	0
2	cija#	32	1	2	restolonaslednikovica$ko...#	0
2	ija#	33	1	11	slednikovica$ko...#	1
2	ja#	34	1	4	stolonaslednikovica$ko...#	0
2	a#	35	1	5	tolonaslednikovica$ko...#	0
2	#	36	1	19	vica$ko...#	0
2		37	2	30	zacija#	

Нека је ST_i парцијално суфиксно стабло добијено уметањем првих i лексикографски најмањих суфикса, $0 \leq i \leq n$. Нека је даље $d(v)$ дужина конкатенације ознака свих грана у стаблу ST_i на путу од корена до чвора v. Полази се од ST_0, стабла које има само корен. Да би се у стабло ST_i убацио суфикс са индексом $SA[i]$, $1 \leq i \leq n$, прелази се пут од последњег уметнутог листа ка корену, до првог чвора v за који је $d(v) \leq LCP[i]$. Могућа су два случаја:

- $d(v) = LCP[i]$ [**уметање нове гране из постојећег чвора**]: тада је дужина конкатенације ознака грана на путу од корена до чвора v једнака $LCP[S_{SA[i-1]}, S_{SA[i]} = LCP[i-1]$. У том случају суфикс са ознаком $SA[i]$ се умеће као нови лист x повезан са чвором v граном (v, x) са ознаком $S_{SA[i]+LCP[i]}$. Другим речима, ознака додате гране се састоји од преосталих знакова суфикса $SA[i]$ који нису део конкатенације ознака грана на путу од корена до чвора v. На тај начин формирано је парцијално суфиксно стабло ST_i.
- $d(v) < LCP[i]$ [**нова грана из чвора уметнутог у постојећу грану**]:

тада је конкатенација ознака грана на путу од корена до чвора v има мање знакова од $LCP[S_{SA[i-1]}, S_{SA[i]}] = LCP[i-1]$ и знакови који недостају до чвора v садржани су на почетку ознаке *најдеснијe* (последње уметнуте) гране (v, w) која излази из чвора v. Због тога се грана (v, w) мора новим унутрашњим чвором y раздвојити на две гране (v, y) и (y, w). При томе гране (v, y) и (y, w) као ознаке добијају одговарајући почетак, односно завршетак ознаке уклоњене гране (v, w). Прецизније,

1. Уклонити грану (v, w).
2. Додати нови унутрашњи чвор y и нову грану (v, y) са ознаком $s[SA[i-1]]+d(v), SA[i-1]+LCP[i]-1]$. Ознака нове гране састоји се од знакова који недостају на најдужем заједничком префиксу суфикса $S_{SA[i-1]}$ и $S_{SA[i]}$. Према томе, конкатенације ознака грана на путу од корена до чвора y једнака је најдужем заједничком префиксу суфикса $S_{SA[i-1]}$ и $S_{SA[i]}$.
3. Повезати чвор w са новокреираним унутрашњим чвором y граном (y, w) са ознаком $s[SA[i-1]] + LCP[i], SA[i-1] + d(v) - 1]$. Нова ознака састоји се од преосталих знакова обрисане гране (v, w) који нису укључени у ознаку гране (v, y).
4. Уметнути чвор са ознаком $SA[i]$ као нови лист x и повезати га са новим унутрашњим чвором y граном (y, x) са ознаком $S_{[SA[i]]+LCP[i]}$. Према томе, ознака гране (y, x) састоји се од преосталих знакова суфикса $S_{SA[i]}$, који нису укључени у конкатенацију ознака грана на путу од корена до чвора v.
5. На тај начин формирано је парцијално суфиксно стабло ST_i.

Пример 4.9. На следећој слици илустрован је овај поступак формирања суфиксног стабла за ниску миссиссиппи. У опису алгоритма претпоставља се да се суфиксно стабло формира слева удесно, тј. да се нови листови додају повезивањем са чвором на тренутно најдеснијој грани. Због једноставнијег цртања стабла, овде се нови листови прикључују стаблу испод најниже гране уместо десно од најдеснијe гране. Другим речима, суфиксно стабло се формира одозго наниже.

i	$SA[i]$	$s[SA[i]..n]$	$LCP[i]$	Суфиксно стабло
1	11	i	1	i 11
2	8	ippi	1	ppi 8
3	5	issippi	4	ssi ippi 5
4	2	ississippi	0	ssippi 2
5	1	mississippi	0	mississippi 1
6	10	pi	1	p i 10
7	9	ppi	0	pi 9
8	7	sippi	2	s i ppi 4
9	4	sissippi	1	ssippi 4
10	6	ssippi	3	si ppi 6
11	3	ssissippi		ssippi 3

Лако је видети да је амортизована сложеност овог алгоритма $O(n)$. Чворови који се пролазе у кораку i на путу ка корену најдеснијим путем у стаблу ST_i (осим последњег чвора v) уклањају се из најдеснијег пута кад се $A[i]$ дода у стабло као нови лист. Ови чворови се никад не пролазе поново у наредним корацима $j > i$. Према томе, укупан број чворова који се пролазе у току извршења алгоритма је највише $2n$.

4.5 Примене суфиксног стабла

Исти проблеми за које смо видели да се могу решавати применом суфиксног низа (тражење речи у тексту, тражење најдуже заједничке подниске две или више ниски, тражење најдужег палиндрома у нисци) могу се још једноставније решавати применом суфиксног стабла.

Глава 5

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.

5.1 Упаривање

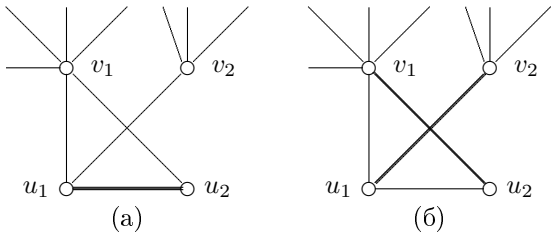
За задати неусмерени граф $G = (V, E)$ **упаривање** је скуп дисјунктних грана (грана без заједничких чворова). Ово име потиче од чињенице да се гране могу схватити као парови чворова. Битно је да сваки чвор припада највише једној грани — то је моногамно упаривање. Чвор који није суседан ни једној грани 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.

5.1.1 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 5.1(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 5.1(б).



Слика 5.1: 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.

5.1.2 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.

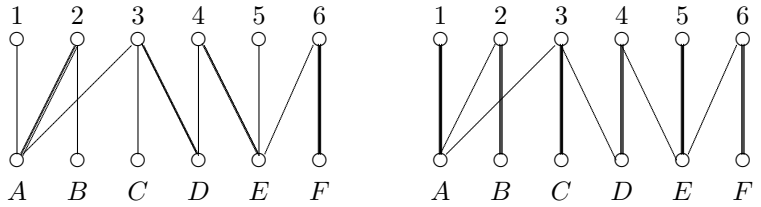
Нека је $G = (V, E, U)$ бипартитни граф у коме су V и U дисјунктни скупови чворова, а E је скуп грана које повезују неке чворове из V са неким чворовима из U.

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 5.2(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 5.2(б).



Слика 5.2: 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.

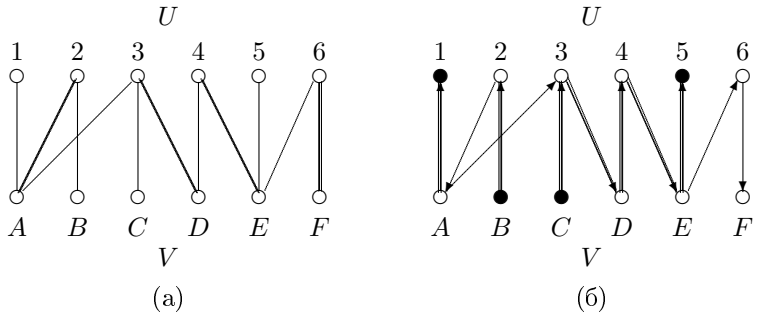
Алтернирајући пут P за дато упаривање M је пут од неупареног чвора $v \in V$ до неупареног чвора $u \in U$, при чему су гране пута P наизменично у $E \setminus M$, односно M. Другим речима, прва грана (v, w) пута P не припада M (јер је v неупарен), друга грана (w, x) припада M, и тако даље до последње 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 $E \setminus M$ 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 5.2(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 1 (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 $E \setminus M$ од V ка U. Слика 5.3(а) приказује полазно максимално упаривање за граф са слике 5.2(а), а слика 5.3(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 5.3: 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 $O((|V|+|E|)\sqrt{|V|})$.

5.2 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;

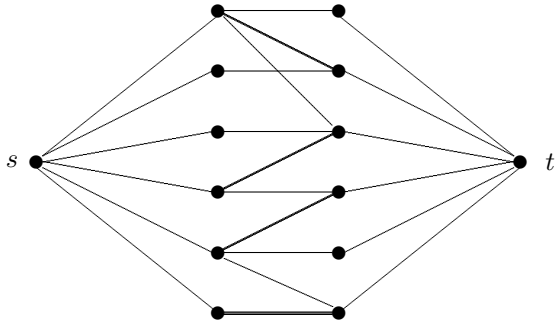
(2) за све чворове $v \in V \setminus \{s, t\}$ је $\sum_u f(u, v) = \sum_w f(v, w)$: укупан ток који улази у произвољни чвор v различит од s, t једнак је укупном току који излази из њега ("нестишљивост", закон очувања, односно конзервације тока).

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 5.4, 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.



Слика 5.4: Свођење бипартитног упаривања на оптимизацију транспортне мреже (све гране усмерене су слева удесно).

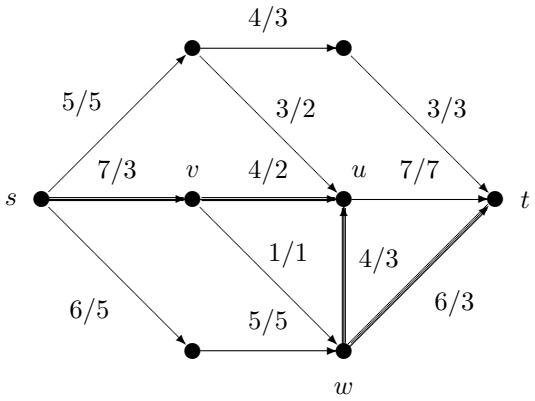
Повећавајући пут у односу на задати ток f је усмерени пут од s до t, који се састоји од грана из G, не обавезно у истом смеру; свака од тих грана (v, u) треба да задовољи тачно један од следећа два услова:

(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".

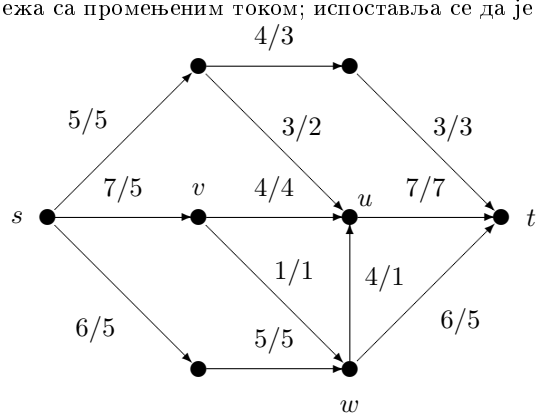
Повећавајући пут је уопштење алтернирајућег пута, и има исти смисао за транспортне мреже као алтернирајући пут за бипартитно упаривање. Ако постоји повећавајући пут у односу на ток f, онда f није оптимални ток. Ток f може се повећати повећавањем тока кроз повећавајући пут на следећи начин. Ако су све 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 5.5. 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.

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



Слика 5.5: Пример мреже са повећавајућим путем.

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 5.6 приказана је иста мрежа са промењеним током; испоставља се да је нови ток у ствари оптималан.



Слика 5.6: Резултат повећавања тока у мрежи са слике 5.5.

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

Теорема 2 (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 3 (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.

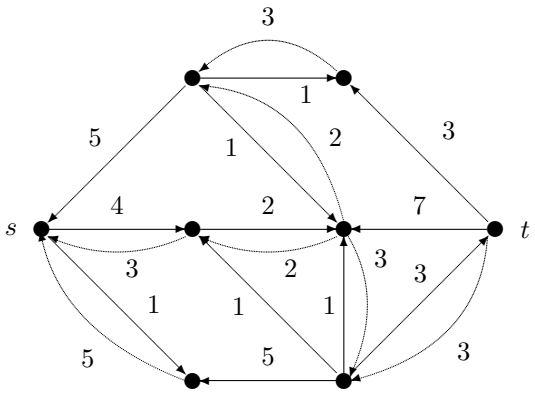
Теорема 4 (Теорема о целобројном току)**.** *Ако су капацитети свих грана у мрежи целобројни, онда постоји оптимални ток са целобројном вредношћу.*

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.

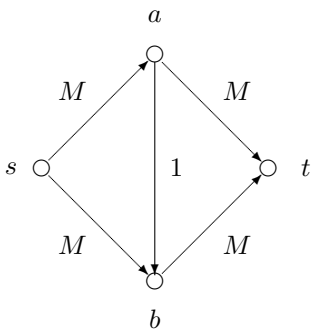
Теорема о повећавајућем путу непосредно се трансформише у алгоритам. Полази се од тока 0, траже се повећавајући путеви, и на основу њих повећава се ток, све до тренутка кад повећавајући путеви више не постоје. Тражење повећавајућих путева може се извести на следећи начин. Дефинишемо **резидуални граф** 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 5.7 приказан је резидуални граф за мрежу са слике 5.5, 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 5.7: Резидуални граф мреже са слике 5.5 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 5.8. 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 5.8: 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

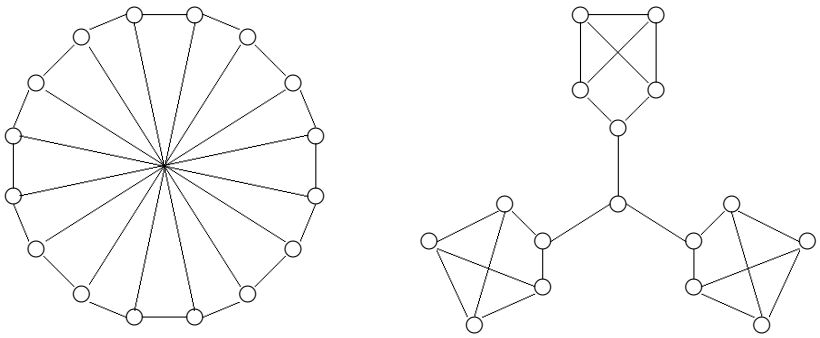
различитих алгоритама, сложенијих и мање сложених, а више њих достиже горњу границу сложености $O(|V|^3)$ у најгорем случају. Овде их нећемо описивати.

5.3 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 5.9 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 5.9: Два графа са по 16 чворова степена 3, од којих је први Хамилтонов, а други није.

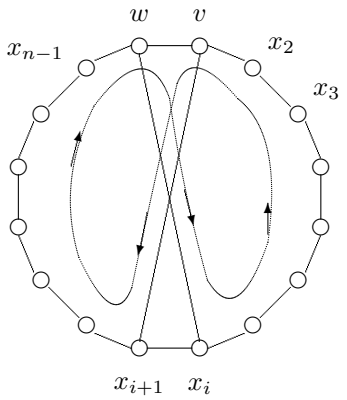
Нека је $G = (V, E)$ повезан неусмерен граф и нека за произвољан чвор $v \in V$ $d(v)$ означава његов степен. Следећи проблем је специјални случај тражења Хамилтоновог циклуса у врло густим графовима. Показаћемо да услови проблема гарантују да је граф Хамилтонов.

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 $x_1, x_2, \ldots, x_n, x_1$ такав циклус у G' (видети слику 5.10). 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 5.10: 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 5.10.

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 6

Редукције

6.1 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 6.4.2 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 6.4.2 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 6.4.2).

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 6.3.

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.

6.2 Примери редукција

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

6.2.1 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.

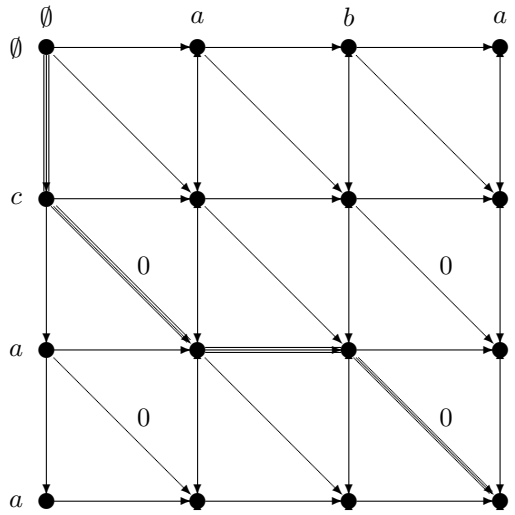
Проблем је установити да ли постоји индекс k, $0 \leq k \leq n-1$, такав да је $a_i = b_{(k+i) \bmod n}$ за све i, $0 \leq i \leq n-1$. Означимо овај проблем са CUS, а основни проблем тражења речи у тексту са TUT. Проблем CUS може се решити, на пример, изменом алгоритма KMP. Постоји бољи начин да се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

6.2.2 Редукције са упоређивањем низова

Размотримо проблем одређивања едит растојања две ниске: дати су низови знакова $A = a_1a_2 \dots a_n$, $B = b_1b_2 \dots b_m$, 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 6.1, где је $A = caa$ и $B = aba$. Хоризонталне гране одговарају уметањима, вертикалне брисањима, а дијагоналне заменама знакова. Тако, на пример, пут истакнут на слици 6.1 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.



Слика 6.1: Граф који одговара низовима $A = caa$ и $B = aba$.

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. 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 $c_1 + c_2 k$, 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.

6.2.3 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 6.4.2, 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.

6.3 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**.

6.3.1 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 $\mathbf{x}$:

$$c(\mathbf{x}) = \sum_{i=1}^{n} c_i x_i, \tag{6.1}$$

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

$$\mathbf{a}_j \cdot \mathbf{x} \leq b_j, \quad 1 \leq j \leq k, \tag{6.2}$$

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

$$\mathbf{e}_j \cdot \mathbf{x} = d_j, \quad 1 \leq j \leq m, \tag{6.3}$$

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)

$$x_j \geq 0 \quad \text{за све } j \in P, \tag{6.4}$$

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

Проблем линеарног програмирања може се формулисати на следећи начин: максимизирати функцију $c(\mathbf{x})$ (6.1), под условом да су задовољена неједнакосна (6.2), једнакосна (6.3) и ненегативна ограничења (6.4). 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.

6.3.2 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 5.2. Нека променљиве $x_1, x_2, \ldots, x_n$ 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 променљиве треба да задовоље ненегативна ограничења $x_i \geq 0$ за $i \in \{1, 2, \ldots, n\}$. Вредности $\mathbf{x}$ које максимизирају циљну 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 5.1.2. Проблем се може решити поступком сличним поступцима за налажење оптималног упаривања, а може се формулисати и као проблем линеарног програмирања. Укупно има nk променљивих 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:

$$\begin{aligned} x_{ij} &\leq a_{ij} \quad \text{за све } i, j, \\ \sum_{j=1}^{k} x_{ij} &\leq s_i \quad \text{за све } i, \\ \sum_{i=1}^{n} x_{ij} &\leq t_j \quad \text{за све } j. \end{aligned}$$

Поред тога, све променљиве 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**.

Променљиве 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 променљиве 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

$$\begin{aligned} \sum_{j=1}^{k} x_{ij} &= 1 \quad \text{за све } i, \\ \sum_{i=1}^{n} x_{ij} &= 1 \quad \text{за све } j. \end{aligned}$$

Поред тога, све променљиве 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. Променљиве треба да представљају избор типа "да"–"не", а њихове оптималне вредности могу да буду реални (дакле не цели) бројеви! Потребно је додати ограничење да променљиве могу узимати само вредности 0 или 1. У општем случају такав проблем је тешко решити. Линеарни програми чије променљиве 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 променљиву 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 променљиве 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 променљивих 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 променљиве 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 промењивих 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 промењиву 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 промењивих $y_k, k = 1, 2, \ldots, n$ којима кодирамо услов да ли се боја k користи ($y_k = 1$) или не ($y_k = 0$). Уводимо и n^2 промењивих 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.

6.4 Примена редукција на налажење доњих граница

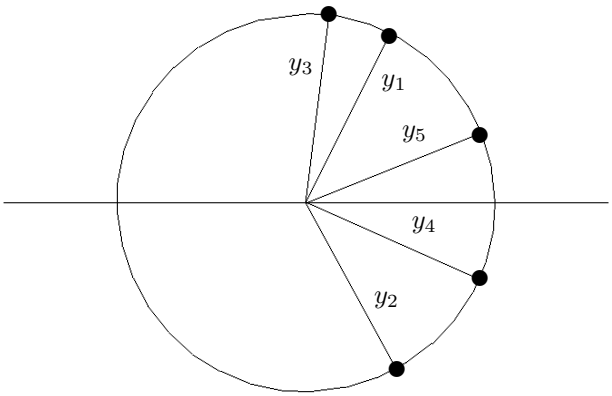
Ако покажемо да се произвољан алгоритам за решавање проблема A може модификовати (не повећавајући му при томе значајно временску сложеност) тако да решава проблем B, онда је доња граница за проблем B истовремено и доња граница за проблем A. Заиста, сваки алгоритам за A је истовремено и алгоритам за B,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

6.4.1 Доња граница за конструкцију простог многоугла

Посматрајмо проблем повезивања скупа тачака у равни простим многоуглом (видети одељак **??**). 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 5. *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 6.2. 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 6.2: 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 6. 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.

6.4.2 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

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} 0 & B^T \\ B & 0 \end{pmatrix} = \begin{pmatrix} AB & 0 \\ 0 & A^T B^T \end{pmatrix}. \tag{6.5}$$

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 7. *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 (6.5). 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 8. *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 7, треба пронаћи матрицу чији квадрат садржи довољно информација за израчунавање производа две задате матрице. Решење се заснива на следећем изразу:

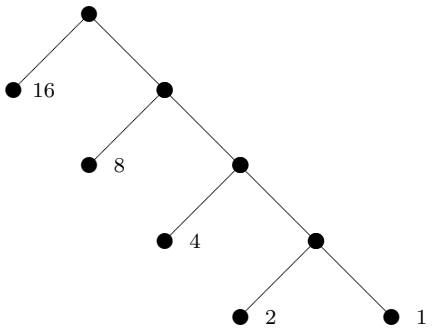
$$\begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}^2 = \begin{pmatrix} AB & 0 \\ 0 & BA \end{pmatrix}.$$

Из њега непосредно следи тврђење теореме.

6.5 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 6.3. 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 $\Omega(n \log n)$.



Слика 6.3: Хафманово кодно стабло кад се учестаности знакова драстично разликују.

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: 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 6.3 (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.

6.6 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 7

NP комплетни проблеми

7.1 Увод

Ово поглавље је битно другачије од осталих. У претходним поглављима разматране су технике за решавање алгоритамских проблема и њихове примене их на конкретне проблеме. Било би лепо кад би сви проблеми имали елегантне ефикасне алгоритме, до којих долази коришћењем малог скупа техника. Међутим, стварни живот није тако једноставан. Постоји много проблема који се не покоравају до сада размотреним техникама. Могуће је да приликом њиховог решавања није уложено довољно напора, али се са доста разлога може претпоставити да постоје проблеми који *немају* ефикасно опште решење. У овом поглављу описаћемо технике за препознавање неких таквих проблема.

Временска сложеност већине до сада разматраних алгоритама ограничена је неким полиномом од величине улаза. За такве алгоритме кажемо да су **ефикасни** алгоритми, а за одговарајуће проблеме да су **решиви**. Другим речима, за алгоритам кажемо да је ефикасан ако је његова временска сложеност $O(P(n))$, где је $P(n)$ полином од величине проблема n. Подсетимо се да је величина улаза дефинисана као број бита потребних за представљање улаза. Ово није строго формална дефиниција: могуће је исте податке представити битима на више начина; међутим, дужине свих довољно ефикасних представљања истих података не могу се значајно разликовати. Класу свих проблема који се могу решити ефикасним алгоритмом означаваћемо са P (полиномијално време). Ова дефиниција може да изгледа чудно на први поглед. Тако, на пример, алгоритам сложености $O(n^{10})$ се ни по којим мерилима не може сматрати ефикасним; слично, алгоритам са временом извршавања $10^7 n$ тешко је сматрати ефикасним, иако је линеарне сложености. Ипак, ова дефиниција има смисла из два разлога. Прво, она омогућује развој теорије којом ћемо се сада позабавити; друго, и важније, ову дефиницију је лако применити. Испоставља се да огромна већина решивих проблема има практично употребљива решења. Другим речима, временска сложеност полиномијалних алгоритама на које се у пракси наилази је најчешће полином малог степена, ретко изнад квадратног. Обрнуто је обично такође тачно: алгоритми са временском сложеношћу већом од произвољног полинома обично се не могу практично извршавати за велике улазе.

Постоји доста проблема за које се не зна ни један алгоритам полиномијалне сложености. Неки од тих проблема ће једном можда бити решени ефикасним алгоритмима. Међутим, има разлога за веровање да се многи проблеми не могу решити ефикасно. Волели бисмо да будемо у стању да препознамо такве проблеме, да не бисмо губили време на тражење непостојећег алгоритма. У овом поглављу

размотрићемо како приступати проблемима за које се не зна да ли су у класи P. Специјално, размотрићемо једну потклasу таквих проблема, класу такозваних NP-**комплетних проблема**. Ови проблеми могу се груписати у једну класу јер су сви међусобно строго еквивалентни — *ефикасни алгоритам за неки* NP-*комплетан проблем постоји ако и само ако за сваки* NP-*комплетан проблем постоји ефикасни алгоритам*. Широко је распрострањено веровање да не постоји ефикасан алгоритам за било који NP-комплетан проблем, али се не зна доказ оваквог тврђења. Чак и кад би постојали ефикасни алгоритми за решавање NP-комплетних проблема, они би сигурно били врло компликовани, јер се таквим проблемима много истраживача бавило током дугог низа година. До овог тренутка се за стотине (можда хиљаде) проблема зна да су NP-комплетни, што ову област чини врло значајном.

Ово поглавље састоји се из два дела. У првом се дефинише класа NP-комплетних проблема и наводе примери доказа припадности неких проблема тој класи. Затим се разматра неколико техника за *приближно* решавање NP-комплетних проблема. Та решења нису увек оптимална, или се не могу применити на произвољан улаз, али је боље имати њих него ништа.

7.2 Редукције полиномијалне временске сложености

У овом одељку ограничићемо се на **проблеме одлучивања**, тј. разматраћемо само оне проблеме на које се после извршавања одређеног алгоритма може одговорити са "да" или "не". Ово ограничење упрошћава разматрања. Већи део проблема лако се може превести у проблеме одлучивања. На пример, уместо да тражимо оптимално упаривање у задатом графу, можемо да поставимо питање да ли за задато k постоји упаривање величине бар k. Ако умемо да решимо проблем одлучивања, обично можемо да решимо и полазни проблем — бинарном претрагом, на пример.

Проблем одлучивања може се посматрати као **проблем препознавања језика**. Нека је U скуп могућих улаза за проблем одлучивања. Нека је $L \subseteq U$ скуп свих улаза за које је решење проблема "да". За L се каже да је **језик** који одговара проблему, па појмови *проблем* и *језик* могу да се користе равноправно. Проблем одлучивања је установити да ли задати улаз припада језику L. Сада ћемо увести појам редукције полиномијалне сложености између језика, као основни алат у овом поглављу.

Дефиниција 1. *Нека су L_1 и L_2 два језика, подскупа редом скупова улаза U_1 и U_2. Кажемо да је L_1* ***полиномијално сводљив*** *на L_2, ако постоји алгоритам полиномијалне временске сложености, који дати улаз $u_1 \in U_1$ преводи у улаз $u_2 \in U_2$, тако да $u_1 \in L_1$ ако и само ако $u_2 \in L_2$. Алгоритам је полиномијалан у односу на величину улаза u_1. Претпостављамо да је појам величине добро дефинисан у просторима улаза U_1 и U_2, тако да је, на пример, величина u_2 ограничена полиномом од величине u_1.*

Алгоритам из дефиниције своди један проблем на други. Ако знамо алгоритам за препознавање L_2, онда га можемо *суперпонирати* са алгоритмом редукције и тако добити алгоритам за решавање L_1. Означимо алгоритам редукције са AR, а алгоритам за препознавање L_2 са AL_2. Произвољни улаз $u_1 \in U_1$ може се применом AR трансформисати у улаз $u_2 \in U_2$, и применом AL_2 установити да ли $u_2 \in L_2$, а тиме и да ли $u_1 \in L_1$. Последица специјалног случаја овог разматрања је следећа теорема.

Теорема 9. *Ако је језик L_1 полиномијално сводљив на језик L_2, и постоји алгоритам полиномијалне временске сложености за препознавање L_2, онда постоји алгоритам полиномијалне временске сложености за препознавање L_1.*

Доказ. Доказ следи из претходног разматрања.

Релација полиномијалне сводљивости није симетрична: полиномијална сводљивост L_1 на L_2 не повлачи увек полиномијалну сводљивост L_2 на L_1. Ова асиметрија потиче од чињенице да дефиниција сводљивости захтева да се *произвољан* улаз за L_1 може трансформисати у еквивалентан улаз за L_2, али не и обрнуто. Могуће је да улази за L_2, добијени на овај начин, представљају само мали део свих могућих улаза за L_2. Према томе, ако је L_1 полиномијално сводљив на L_2, онда можемо сматрати да је проблем L_2 *тежи*.

Два језика L_1 и L_2 су **полиномијално еквивалентни**, или једноставно еквивалентни, ако је сваки од њих полиномијално сводљив на други. Специјално, сви нетривијални проблеми из класе P су еквивалентни. Релација "полиномијалне сводљивости" је транзитивна, као што показује следећа теорема.

Теорема 10. *Ако је L_1 полиномијално сводљив на L_2 и L_2 је полиномијално сводљив на L_3, онда је L_1 полиномијално сводљив на L_3.*

Доказ. Нека су језици L_1, L_2 и L_3 подскупови скупова могућих улаза редом U_1, U_2 и U_3. Суперпонирањем алгоритама редукције L_1 на L_2, односно L_2 на L_3 добија се алгоритам редукције L_1 на L_3. Произвољан улаз $u_1 \in U_1$ конвертује се најпре у улаз $u_2 \in U_2$, који се затим конвертује у улаз $u_3 \in U_3$. Пошто су редукције полиномијалне сложености, а композиција две полиномијалне функције је полиномијална функција, резултујући алгоритам редукције је такође полиномијалне сложености (то је један од разлога зашто су за меру сложености решивих проблема изабрани полиноми).

Суштина метода који ћемо примењивати у овом поглављу је да ако се за неки проблем не може наћи ефикасан алгоритам, онда покушавамо да установимо да ли је он еквивалентан неком од проблема за које знамо да су тешки. Класа NP-комплетних проблема садржи стотине таквих међусобно еквивалентних проблема.

7.3 Недетерминизам и Кукова теорема

Теорија NP-комплетности заснована је значајном Куковом (S. A. Cook) теоремом 1971. године. Пре него што формулишемо ову теорему, потребно је да уведемо још неке појмове. Покушаћемо да избегнемо техничке детаље и да дискусију задржимо на интуитивном нивоу. Књига Герија и Џонсона [?] (M. R. Garey, D. S. Johnson) добро покрива ову област. Теорија NP-комплетности је део шире области која се зове **сложеност израчунавања**, чији највећи део није предмет ове књиге. Разматрање ћемо ограничити на оне делове који омогућују коришћење те теорије.

Сада ћемо дефинисати другу важну класу језика (односно проблема одлучивања) — класу NP. Пре него што пређемо на формалну дефиницију засновану на Тјуринговим машинама, покушаћемо да разјаснимо смисао појмова на којима се заснива класа NP. Размотримо као пример проблем трговачког путника.

Проблем. Задат је тежински граф $G = (V, E)$ са $|V| = n$ чворова (тако да је за произвољне чворове – градове $v_i, v_j \in V$, тежина гране $(v_i, v_j) \in E$ једнака $d(v_i, v_j)$) и број $B \in \mathbf{Z}^+$. Установити да ли у G постоји Хамилтонов циклус са збиром тежина грана мањим или једнаким од B. Другим речима, постоји ли такав низ чворова $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$ да је

$$\sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)}) + d(v_{\pi(n)}, v_{\pi(1)}) \leq B?$$

Полиномијални алгоритам за решавање овог проблема није познат. Претпоставимо, међутим, да је за неки конкретан улаз за овај проблем добијен одговор "да". Ако у то посумњамо, можемо захтевати "доказ" тог тврђења — низ чворова који има тражену особину. Кад имамо овакав низ чворова, лако можемо да проверимо да ли је то Хамилтонов циклус, да израчунамо његову дужину и упоредимо је са задатом границом B. Поред тога, очигледно је временска сложеност оваквог алгоритма провере полиномијална у односу на величину улаза.

Управо појам полиномијалне проверљивости карактерише класу NP. Приметимо да проверљивост за полиномијално време не повлачи и могућност решавања за полиномијално време. Утврђујући да се за полиномијално време може проверити одговор "да" за проблем трговачког путника, ми не узимамо у обзир време које нам може бити потребно за проналажење жељеног циклуса у експоненцијалном броју свих могућих циклуса у графу. Ми само тврдимо да се за сваки задати низ чворова и за конкретан улаз u, за полиномијално време може проверити да ли тај низ "доказује" да је за улаз u одговор "да".

Класа NP се неформално може дефинисати помоћу појма *недетерминистичког алгоритма*. Такав алгоритам састоји се од две различите фазе: *фазе погађања* и *фазе провере*. За задати улаз u, у првој фази се изводи просто "погађање" неке структуре S. Затим се u и S заједно предају као улаз фази провере, која се изводи на обичан (детерминистички) начин, па се завршава одговором "да" или "не", или се извршава бесконачно дуго. Недетерминистички алгоритам "решава" проблем одлучивања П, ако су за произвољни улаз $u \in U_{\text{П}}$ за овај проблем испуњена следећа два услова:

1. Ако $u \in L_{\text{П}}$, онда постоји таква структура S, чије би погађање за улаз u довело до тога да се фаза провере са улазом (u, S) заврши одговором "да".
2. Ако $u \notin L_{\text{П}}$, онда *не* постоји таква структура S, чије би погађање за улаз u обезбедило завршавање фазе провере са улазом (u, S) одговором "да".

На пример, недетерминистички алгоритам за решавање проблема трговачког путника могао би се конструисати користећи као фазу погађања избор произвољног низа градова (чворова), а као фазу провере — горе поменути алгоритам "провере доказа" за проблем трговачког путника. Очигледно је да за произвољан конкретан улаз u постоји такво погађање S, да се као резултат рада фазе провере са улазом (u, S) добија "да", онда и само онда, ако за улаз u постоји Хамилтонов циклус тражене дужине.

Каже се да недетерминистички алгоритам који решава проблем одлучивања П ради за "полиномијално време", ако постоји полином p такав да за сваки улаз $u \in U_{\text{П}}$ постоји такво погађање S, да се фаза провере са улазом (u, S) завршава са одговором "да" за време $p(|u|)$. Из тога следи да је величина структуре S обавезно ограничена полиномом од величине улаза, јер се на проверу погађања S може утрошити највише полиномијално време.

Класа NP, дефинисана неформално, — то је класа свих проблема одлучивања који при разумном кодирању могу бити решени недетерминистичким (N – nondeterministic) алгоритмом за полиномијално (P – polynomial) време. На пример, проблем трговачког путника припада класи NP.

У сличним неформалним дефиницијама термин "решава" треба опрезно користити. Треба да буде јасно да је основни смисао "полиномијалног недетерминистичког алгоритма" у томе да објасни појам "проверљивости за полиномијално време", а не у томе да буде реални метод решавања проблема одлучивања. За сваки конкретан улаз такав алгоритам има не једно, него неколико могућих извршавања — по једно за свако могуће погађање.

Недетерминистички алгоритми су врло моћни, али њихова снага није неограничена. Постоје проблеми који се не могу ефикасно решити недетерминистичким алгоритмом. На пример, посматрајмо следећи проблем: да ли је величина

оптималног упаривања у задатом графу једнака тачно k? Недетерминистичким алгоритмом лако се може пронаћи упаривање величине k, ако оно постоји, али се не може лако установити (чак ни недетерминистички) да не постоји веће упаривање.

Класа проблема за које постоји недетерминистички алгоритам полиномијалне временске сложености зове се NP. Изгледа разумно сматрати да су недетерминистички алгоритми моћнији од детерминистичких, али да ли је то тачно? Један начин да се то докаже био би да се пронађе неки NP проблем који није у P. То до сада никоме није пошло за руком. У противном, да би се доказало да су ове две класе једнаке (односно P=NP), требало би показати да сваки проблем из класе NP може да буде решен детерминистичким алгоритмом полиномијалне временске сложености. Ни овакво тврђење нико није успео да докаже (и мало је оних који верују у његову тачност). Проблем утврђивања односа између P и NP познат је као **проблем** P=NP.

Сада ћемо дефинисати две класе, које не само да садрже бројне важне проблеме за које се не зна да ли су у P, него садрже **најтеже** проблеме у NP.

Дефиниција 2. *Проблем X је* NP-***тежак проблем*** *ако је сваки проблем из класе* NP *полиномијално сводљив на X.*

Дефиниција 3. *Проблем X је* NP-***комплетан проблем*** *ако (1) припада класи* NP*, и (2) X је* NP*-тежак. Класу* NP*-комплетних проблема означаваћемо са* NPC*.*

Последице дефиниције класе NP-тешких проблема је да, ако се за било који NP-тежак проблем докаже да припада класи P, онда је P=NP.

Кук је 1971. године доказао да *постоје* NP-комплетни проблеми, односно да класа NPC није празна. Специјално, навео је пример једног таквог проблема, који ћемо ускоро приказати. Када се зна један NP-комплетан проблем, докази да су други проблеми у класи NPC постају једноставнији. Ако је дат нови проблем X, довољно је доказати да је Куков проблем (или било који други NP-комплетан проблем) полиномијално сводљив на X. То следи из следеће леме.

Лема 7.1. Проблем X је NP-комплетан ако (1) X припада класи NP, и (2') постоји NP-комплетан проблем Y који је полиномијално сводљив на X.

Доказ. Проблем Y је према услову (2) NP-тежак, па је сваки проблем у класи NP полиномијално сводљив на Y. Пошто је Y полиномијално сводљив на X, а полиномијална сводљивост је транзитивна релација, сваки проблем у класи NP је полиномијално сводљив и на проблем X.

Много је лакше доказати да су два проблема полиномијално сводљива, него директно доказати да је испуњен услов (2). Због тога је Куков резултат камен темељац целе теорије. Даље, са појавом нових проблема за које се зна да су NP-комплетни, расте број могућности за доказивање услова (2'). Убрзо пошто је Куков резултат постао познат, Карп (R. M. Karp) је за 24 важна проблема показао да су NP-комплетни. Од тог времена је за стотине проблема (можда хиљаде, зависно од начина бројања варијација истог проблема) доказано да су NP-комплетни. У следећем одељку приказаћемо шест примера NP-комплетних проблема, са доказима њихове NP-комплетности. Навешћемо такође (без доказа) више NP-комплетних проблема. Најтежи део доказа је обично (не увек) доказ испуњености услова (2').

Изложићемо сада проблем за који је Кук доказао да је NP-комплетан, са идејом доказа. Проблем је познат као **проблем задовољивости** (SAT, скраћеница од satisfiability). Нека је S Булов израз у **конјуктивној нормалној форми** (КНФ). Другим речима, S је конјункција више *клауза* (дисјункција

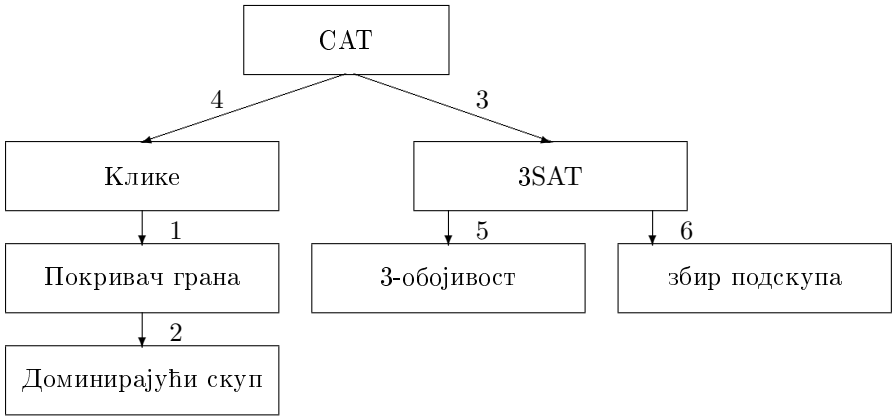
група *литерала* — симбола променљивих или њихових негација). На пример, $S = (x+y+\bar{z})\cdot(\bar{x}+y+z)\cdot(\bar{x}+\bar{y}+\bar{z})$, где сабирање, односно множење одговарају дисјункцији, односно конјункцији (*или*, односно *и*), а свака променљива има вредност 0 (нетачно) или 1 (тачно). Познато је да се свака Булова функција може представити изразом у КНФ. За Булов израз се каже да је **задовољив**, ако постоји такво додељивање нула и јединица променљивим, да израз има вредност 1. Проблем SAT састоји се у утврђивању да ли је задати израз задовољив (при чему није неопходно пронаћи одговарајуће вредности променљивих). У наведеном примеру израз S је задовољив, јер за $x = 1$, $y = 1$ и $z = 0$ има вредност $S = 1$.

Проблем SAT је у класи NP јер се за (недетерминистички) изабране вредности променљивих за полиномијално време (од величине улаза — укупне дужине формуле) може проверити да ли је израз тачан. Доказ да је проблем SAT NP-тежак овде изостављамо.

Теорема 11 (Кукова теорема)**.** *Проблем* SAT *је* NP*-комплетан.*

7.4 Примери доказа NP-комплетности

У овом одељку доказаћемо да су NP-комплетни следећих пет проблема: покривач грана, доминирајући скуп, 3SAT, 3-обојивост и проблем клика. Сваки од ових проблема биће детаљније изложен. Доказивање NP-комплетности заснива се на техникама које ће бити резимиране на крају одељка. Да би се доказала NP-комплетност неког проблема, мора се најпре доказати да он припада класи NP, што је обично (али не увек!) лако, а затим треба пронаћи редукцију полиномијалне временске сложености на наш проблем неког проблема за који се зна да је NP-комплетан. Редослед редукција у доказима NP-комплетности шест наведених проблема приказан је на слици 7.1. Да би се ови докази лакше разумели, наведени су редоследом према тежини, а не према растојању од корена стабла на слици 7.1; редослед доказа приказан је бројевима уз гране.



Слика 7.1: Редослед доказа NP-комплетности у тексту.

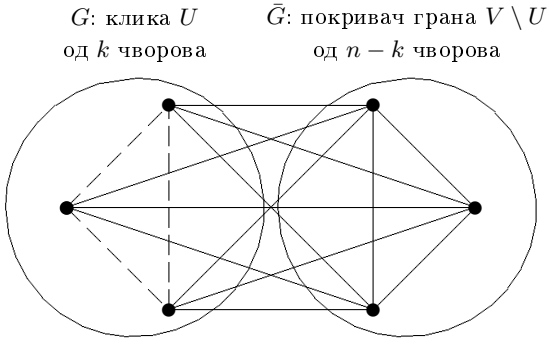
7.4.1 Покривач грана

Нека је $G = (V, E)$ неусмерени граф. **Покривач грана** графа G је такав скуп чворова, да је свака грана G суседна бар једном од чворова из скупа.

Проблем. Задат је неусмерени граф $G = (V, E)$ и природни број k. Установити да ли у G постоји покривач грана са $\leq k$ чворова.

Теорема 12. *Проблем покривач грана је* NP*-комплетан.*

Доказ. Проблем покривач грана је у класи NP, јер се за претпостављени подскуп од $\leq k$ чворова лако проверава за полиномијално време да ли је покривач грана графа. Да бисмо доказали да је проблем покривач грана NP-комплетан, треба да на њега сведемо неки NP-комплетан проблем. У ту сврху искористићемо проблем клика (доказ да је проблем клика NP-комплетан биће дат у одељку 7.4.4). У неусмереном графу $G = (V, E)$ **клика** је такав подграф C графа G у коме су сви чворови међусобно повезани гранама из G. Другим речима, клика је комплетан подграф. Проблем клика гласи: за задати граф G и природни број k, установити да ли G садржи клику величине k. Потребно је трансформисати произвољан улаз за проблем клика у улаз за проблем покривач грана, тако да је решење проблема клика "да" акко је "да" решење одговарајућег проблема покривач грана. Нека је $G = (V, E)$, k произвољан улаз за проблем клика. Нека је $\bar{G} = (V, \bar{E})$ **комплемент графа** G, односно граф са истим скупом чворова као и G, и комплементарним скупом грана у односу на G (односно између произвољна два чвора у $\bar{G}$ грана постоји акко између та два чвора у G *не* постоји грана). Нека је $n = |V|$. Тврдимо да је проблем клика (G, k) сведен на проблем покривач грана графа, са улазом $\bar{G}$, $n-k$ (видети пример на слици 7.2, где су испрекиданим линијама приказане гране из G). Претпоставимо да је $C = (U, F)$ клика у G. Скуп чворова $V \setminus U$ покрива све гране у $\bar{G}$, јер у $\bar{G}$ нема грана које повезују чворове из U (све те гране су у G). Према томе, $V \setminus U$ је покривач грана за $\bar{G}$. Другим речима, ако G има клику величине k, онда $\bar{G}$ има покривач грана величине $n-k$. Обрнуто, нека је D покривач грана у $\bar{G}$. Тада D покрива све гране у $\bar{G}$, па у $\bar{G}$ не може да постоји грана која повезује нека два чвора из $V \setminus D$. Према томе, $V \setminus D$ је клика у G. Закључујемо да ако у $\bar{G}$ постоји покривач грана величине $n-k$, онда у G постоји клика величине k. Ова редукција се очигледно може извршити за полиномијално време: потребно је само конструисати граф $\bar{G}$ полазећи од графа G и израчунати разлику $n - k$.



Слика 7.2: Редукција проблема клика на проблем покривач грана.

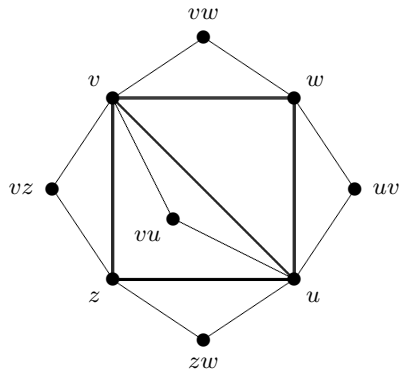
7.4.2 Доминирајући скуп

Нека је $G = (V, E)$ неусмерени граф. **Доминирајући скуп** је скуп $D \subset V$, такав да је сваки чвор G у D, или је суседан бар једном чвору из D.

Проблем. Дат је неусмерени граф $G = (V, E)$ и природни број k. Установити да ли у G постоји доминирајући скуп са највише k чворова.

Теорема 13. *Проблем доминирајући скуп је* NP*-комплетан.*

Доказ. Проблем доминирајући скуп је у класи NP, јер се за претпостављени подскуп од највише k чворова лако за полиномијално време проверава да ли је доминирајући скуп. Извешћемо редукцију проблема покривач грана на проблем доминирајући скуп. Ако је задат произвољан улаз (G, k) за проблем покривач грана, циљ је конструисати нови граф G' који има доминирајући скуп одређене величине акко G има покривач грана величине највише k. При томе се, без смањења општости, може претпоставити да G нема изолованих чворова (они не утичу на покривач грана, али морају бити укључени у доминирајући скуп). Полазећи од графа G, додајемо му $|E|$ нових чворова и $2|E|$ нових грана на следећи начин. За сваку грану (v, w) из G додајемо нови чвор vw и две нове гране (v, vw) и (w, vw) (слика 7.3). Другим речима, сваку грану трансформишемо у троугао. Означимо нови граф са G'. Граф G' је лако конструисати за полиномијално време.



Слика 7.3: Редукција проблема покривач грана на проблем доминирајући скуп.

Тврдимо да G има покривач грана величине m акко G' има доминирајући скуп величине m. Нека је D доминирајући скуп графа G'. Ако D садржи било који од нових чворова vw, онда се такав чвор може заменити било чвором v, било чвором w, после чега ће и нови скуп бити доминирајући скуп (v и w покривају све чворове које покрива vw). Према томе, без смањења општости може се претпоставити да D садржи само чворове из G. Међутим, пошто D "доминира" над свим новим чворовима, он мора за сваку грану из G да садржи бар један њен крај, па је због тога D истовремено и покривач грана графа G. Обрнуто, ако је C покривач грана у G, онда је свака грана G суседна неком чвору из C, па је и сваки нови чвор из G' суседан неком чвору из C. Стари чворови су такође покривени чворовима из C, јер по претпоставци чворови из C покривају све гране.

7.4.3 3SAT

Проблем 3SAT је упрошћена верзија обичног проблема SAT. Улаз за проблем 3SAT је КНФ у којој свака клауза има тачно три литерала.

Проблем. Задат је Булов израз у КНФ, у коме свака клауза садржи тачно три литерала. Установити да ли је израз задовољив.

Теорема 14. *Проблем* 3SAT *је* NP*-комплетан.*

Доказ. Овај проблем је на први поглед лакши од обичног проблема SAT, због допунског ограничења да свака клауза има по три променљиве. Показаћемо да алгоритам који решава 3SAT може да се искористи да реши обичан проблем SAT (односно да се SAT може свести на 3SAT). Пре тога, јасно је да 3SAT припада класи NP. Могу се изабрати ("погодити") вредности променљивих и за полиномијално време проверити да ли је израз тачан. Нека је E произвољан улаз за SAT. Сваку клаузу у E заменићемо са неколико клауза од по тачно три литерала. Нека је $C = (x_1 + x_2 + \cdots + x_k)$ произвољна клауза из E, таква да је $k \geq 4$. Овде је због удобности са x_i означен литерал, односно било променљива, било негација променљиве. Показаћемо како се C може еквивалентно заменити са неколико клауза од по тачно три литерала. Идеја је увести нове променљиве $y_1, y_2, \ldots, y_{k-3}$, које клаузу трансформишу у део улаза за 3SAT, не мењајући задовољивост израза. За сваку клаузу из E уводе се нове, различите променљиве; C се замењује конјункцијом клауза C', тако да је

$$C' = (x_1+x_2+y_1)(x_3+\bar{y}_1+y_2)(x_4+\bar{y}_2+y_3)\cdots(x_{k-2}+\bar{y}_{k-4}+y_{k-3})(x_{k-1}+x_k+\bar{y}_{k-3}).$$

Тврдимо да је израз, добијен од E заменом C са C', задовољив акко је задовољив израз E. Ако је израз E задовољив, онда бар један од литерала x_i мора имати вредност 1. У том случају се могу изабрати вредности променљивих y_i у C' тако да све клаузе у C' буду тачне. На пример, ако је $x_3 = 1$, онда се може ставити $y_1 = 1$ (што чини тачном прву клаузу), $y_2 = 0$ (друга клауза је тачна због $x_3 = 1$), и $y_i = 0$ за све $i > 2$. Уопште, ако је $x_i = 1$, онда стављамо $y_1 = y_2 = \cdots = y_{i-2} = 1$ и $y_{i-1} = y_i = \cdots = y_{k-3} = 0$, што обезбеђује да буде $C' = 1$. Обрнуто, ако израз C' има вредност 1, тврдимо да бар један од литерала x_i мора имати вредност 1. Заиста, ако би сви литерали x_i имали вредност 0, онда би израз C' имао исту тачност као и израз $C'' = (y_1) \cdot (\bar{y}_1 + y_2) \cdot (\bar{y}_2 + y_3) \cdots (\bar{y}_{k-4} + y_{k-3}) \cdot (\bar{y}_{k-3})$, који очигледно није задовољив (да би било $C'' = 1$, морало би да буде редом $y_1 = 1$, па $y_2 = 1$, итд, $y_{k-3} = 1$, $y_{k-3} = 0$, и $y_{k-3} = 1$, што је контрадикција).

Помоћу ове редукције све клаузе са више од три литерала могу се заменити са неколико клауза од по тачно три литерала. Остаје да се трансформишу клаузе са једним или два литерала. Клауза облика $C = (x_1+x_2)$ замењује се еквивалентном изразом

$$C' = (x_1 + x_2 + z)(x_1 + x_2 + \bar{z}),$$

где је z нова променљива. Коначно, клауза облика $C = x_1$ може се заменити изразом

$$C' = (x_1 + y + z)(x_1 + \bar{y} + z)(x_1 + y + \bar{z})(x_1 + \bar{y} + \bar{z}),$$

у коме су y и z нове променљиве.

Према томе, произвољни улаз за проблем SAT може се свести на улаз за проблем 3SAT, тако да је први израз задовољив акко је задовољив други. Јасно је да се ова редукција изводи за полиномијално време.

7.4.4 Клике

Проблем клика дефинисан је у одељку 7.4.1, у коме је разматран проблем покривач грана.

Проблем. Дат је неусмерени граф $G = (V, E)$ и природни број k. Установити да ли G садржи клику величине бар k.

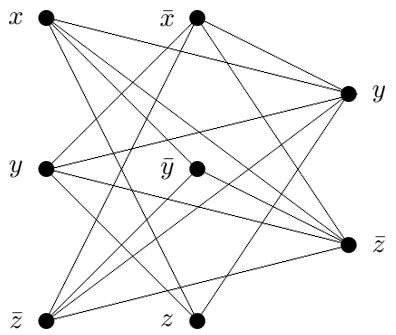
Теорема 15. *Проблем клика је* NP*-комплетан.*

Доказ. Проблем клика је у класи NP, јер се за сваки претпостављени подскуп од k чворова за полиномијално време може проверити да ли је клика. Показаћемо сада да се проблем SAT може свести на проблем клика. Нека је E произвољни Булов израз у КНФ, $E = E_1 \cdot E_2 \cdots E_m$. Посматрајмо, на пример, клаузу $E_i = (x+y+z+w)$. Њој придружујемо "колону" од четири чвора, означена литералима из E_i (без обзира што се неки од њих можда појављују и у другим клаузама). Другим речима, граф G који конструишемо имаће по један чвор за сваку појаву било које променљиве. Остаје питање како повезати ове чворове, тако да G садржи клику величине бар k акко је израз E задовољив. Приметимо да се вредност k може изабрати произвољно, јер је потребно свести проблем SAT на проблем клика, односно решити проблем SAT користећи алгоритам за решавање проблема клика. Наравно, алгоритам за решавање проблема клика мора да ради за сваку вредност k. Ово је важна флексибилност, која се често користи у доказима NP-комплетности. У овом случају за k ћемо изабрати вредност једнаку броју клауза m.

Гране у графу G могу се задати на следећи начин. Чворови из исте колоне (односно чворови придружени литералима из исте клаузе) не повезују се гранама. Чворови из различитих колона су скоро увек повезани: изузетак је случај два чвора од којих један одговара променљивој, а други комплементу те исте променљиве. Пример графа који одговара изразу

$$E = (x + y + \bar{z}) \cdot (\bar{x} + \bar{y} + z) \cdot (y + \bar{z})$$

приказан је на слици 7.4. Јасно је да се G може конструисати за полиномијално време.



Слика 7.4: Редукција проблема SAT са улазом $(x+y+\bar{z})\cdot(\bar{x}+\bar{y}+z)\cdot(y+\bar{z})$ на проблем клика.

Тврдимо да G има клику величине бар m, акко је израз E задовољив. Најпре запажамо да због конструкције максимална клика не може имати више од m чворова, независно од E. Претпоставимо да је израз E задовољив. Тада постоји такво додељивање вредности променљивим, да у свакој клаузи постоји бар један литерал са вредношћу 1. Чвор који одговара том литералу прикључује се клики (ако има више таквих литерала, бира се произвољан од њих). Добијени подграф јесте клика, јер једини начин да два чвора из различитих колона не буду повезана је да један од њих буде комплемент другог — што је немогуће, јер је свим изабраним литералима додељена вредност 1. Обрнуто, претпоставимо да G садржи клику величине бар m. Клика мора да садржи тачно један чвор из сваке колоне (јер по конструкцији чворови из исте колоне нису повезани).

Одговарајућим литералима додељујемо вредност 1. Ако на овај начин некој променљивој није додељена вредност, то се може учинити на произвољан начин. Извдено додељивање вредности променљивима је непротивречно: ако би некој променљивој x и њеном комплементу $\bar{x}$, укљученим у клику, истовремено била додељена вредност 1, то би значило да одговарајући чворови (по конструкцији) нису повезани — супротно претпоставци да су оба чвора у клики.

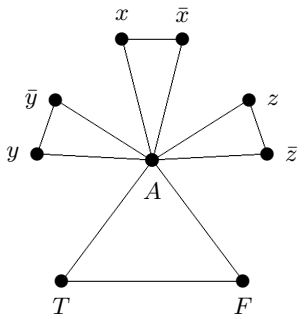
7.4.5 3-обојивост

Нека је $G = (V, E)$ неусмерени граф. **Исправно бојење** (или само бојење) графа G је такво придруживање боја чворовима, да је сваком чвору придружена нека боја, а да су суседним чворовима увек придружене различите боје.

Проблем (3-обојивост). Дат је неусмерени граф $G = (V, E)$. Установити да ли се G може обојити са три боје.

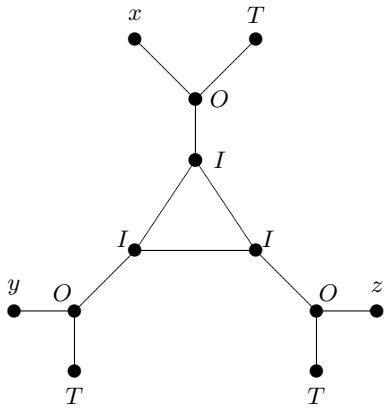
Теорема 16. *Проблем 3-обојивост је* NP*-комплетан.*

Доказ. Проблем 3-обојивост припада класи NP, јер се може претпоставити произвољно бојење графа са 3 боје, а затим за полиномијално време проверити да ли је претпостављено бојење исправно. Извешћемо редукцију проблема 3SAT на проблем 3-обојивост. Доказ је нешто компликованији из два разлога. Најпре, проблеми се односе на различите објекте — Булове изразе у КНФ, односно графове. Друго, не може се просто заменити један објекат (на пример чвор или грана) другим (на пример клаузом); мора се водити рачуна о комплетној структури. Идеја је да се искористе саставни елементи, који се повезују у целину. Нека је E произвољан улаз за проблем 3SAT. Треба конструисати граф G, тако да је израз E задовољив акко је G 3-обојив. Најпре конструишемо *основни троугао* M. Пошто је M троугао (комплетни граф са три чвора), за његово бојење потребне су тачно три боје. Означимо те боје са T (тачно), F (нетачно) и A, видети доњи троугао на слици 7.5. Поред тога, за сваку променљиву x додајемо нови троугао M_x, чије чворове означавамо са x, $\bar{x}$ и A, при чему се чвор означен са A поклапа са чвором из M са истом ознаком. Према томе, ако се у изразу појављује k променљивих, имамо $k + 1$ троуглова са заједничким чвором A (слика 7.5). Ово обезбеђује да, ако је, на пример, чвор x обојен бојом T, онда чвор $\bar{x}$ мора бити обојен бојом F (јер су оба чвора суседна чвору обојеном бојом A), и обрнуто. Ово је у складу са значењем $\bar{x}$.

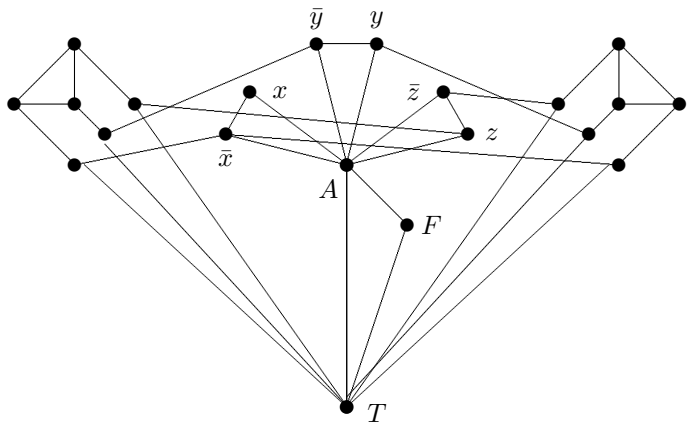


Слика 7.5: Први део конструкције за редукцију 3SAT на 3-обојивост графа.

Потребно је додати услов, који би обезбеђивао да у свакој клаузи бар један литерал има вредност 1. То се може обезбедити на следећи начин. Претпоставимо, на пример, да се ради о клаузи $(x + y + z)$. Овде се x, y, z могу сматрати литералима, тј. недостатак негација над симболима x, y, z не смањује општост разматрања. За ову клаузу уводимо шест нових чворова и повезујемо их са постојећим чворовима на начин приказан на слици 7.6. Назовимо три нова чвора повезана са T и x, y или z *спољашњим чворовима* (означени су са O на слици), а три нова чвора у троуглу — *унутрашњим чворовима* (означени су са I на слици). Тврдимо да ова конструкција обезбеђује (ако је могуће бојење са три боје) да бар један од чворова x, y или z мора бити обојен бојом T. Ни један од чворова x, y, z не може бити обојен бојом A, јер су сви они повезани са чвором A (видети слику 7.5). Ако би сва три чвора x, y, z били обојени бојом F, онда би три нова спољашња чвора повезана са њима морали бити обојени бојом A, па се унутрашњи троугао не би могао обојити са три боје! Комплетан граф који одговара изразу $(\bar{x} + y + \bar{z}) \cdot (\bar{x} + \bar{y} + z)$ приказан је на слици 7.7.



Слика 7.6: Подграфови који одговарају клаузама у редукцији 3SAT на 3-обојивост.



Слика 7.7: Граф који одговара изразу $(\bar{x} + y + \bar{z}) \cdot (\bar{x} + \bar{y} + z)$ у редукцији 3SAT на 3-обојивост.

Сада можемо да комплетирамо доказ, и то у два смера: (1) ако је израз E задовољив, G се може обојити са три боје, и (2) ако се G може обојити са три боје, онда је израз E задовољив. Ако је израз E задовољив, онда постоји такво додељивање вредности променљивим, да у свакој клаузи бар један литерал има вредност 1. Обојимо чворове графа у складу са њиховим вредностима (са T ако је вредност 1, односно са F у противном). Троугао M обојен је бојама T, F и A на описани начин. У подграфу који одговара клаузи бар један литерал има вредност 1; одговарајући спољашњи чвор бојимо са F, а остале спољашње чворове са A, после чега је бојење унутрашњих чворова лако извести. Према томе, G се може обојити са три боје. Обрнуто, ако се G може обојити са три боје, назовимо боје у складу са бојењем троугла M (који мора бити обојен са три боје). Због троуглова на слици 7.5, боје променљивих омогућују непротивречно додељивање вредности променљивим. Конструкција са слике 7.6 обезбеђује да је бар један литерал у свакој клаузи обојен са T. Коначно, јасно је да се граф G може конструисати за полиномијално време, чиме је доказ завршен.

7.4.6 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 7.1 (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.

		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 7.8: 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.

Конструишемо скуп S и циљну вредност t на следећи начин (види слику 7.8). 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 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.

7.4.7 Општа запажања

Размотрићемо укратко неколико општих метода за доказивање NP-комплетности неког проблема Q. Први услов — да Q припада класи NP — обично се лако доказује (не увек). После тога треба изабрати неки проблем за који се зна да је NP-комплетан, а за који изгледа да је повезан, или сличан са Q. Тешко је дефинисати ову "сличност", јер понекад изабрани проблем и Q изгледају јако

различито (на пример, проблем клика и SAT). Избор правог проблема који ће бити сведен на Q је понекад изузетно тежак, и захтева велико искуство. Мора се покушати са неколико редукција, док се не дође до погодног проблема.

Важно је да се изведе редукција на Q, полазећи од *произвољног улаза* познатог NP-комплетног проблема. Најчешћа грешка у оваквим доказима је извођење редукције у обрнутом смеру. Прави смер обезбеђује нпр. следеће правило: треба обезбедити да се познати NP-комплетан проблем може решити црном кутијом, која извршава алгоритам за решавање Q. То можда мало противречи интуицији. Природно би било, покушати са решавањем проблема Q, пошто је то задати проблем. Овде пак покушавамо да решимо други проблем (познати NP-комплетан проблем), користећи решење проблема Q. Ми и не покушавамо да решимо Q!

Постоји више "степени слободе" који се могу користити при редукцији. На пример, ако Q садржи неки параметар, онда се његова вредност може фиксирати на произвољан начин (супротно од параметара у проблему који се своди на Q, који се не смеју фиксирати!). Пошто је Q само алат за решавање познатог NP-комплетног проблема, може се искористити на произвољан начин. Поред фиксирања параметара, могу се користити рестрикције Q на специјалне случајеве добијене и на друге начине. На пример, могу се користити само неки типови улаза за Q (ако се ради о графовима — бипартитни графови, стабла и слично). Други важан извор флексибилности је чињеница да је ефикасност редукције небитна — довољно је да се редукција може извести за полиномијално време. Могу се игнорисати не само константе, тако што би се, на пример, дуплирала величина проблема: исто тако може се и квадрирати величина проблема! Може се увести полиномијално много нових променљивих, може се заменити сваки чвор у графу новим великим графом и слично. Не постоји потреба за ефикасношћу (све док се остаје у границама полиномијалног), јер сврха редукције није да се трансформише у алгоритам.

Постоје неке уобичајене технике за добијање редукција. Најједноставнија је доказати да је познати NP-комплетан проблем специјалан случај проблема Q. Ако је тако, доказ је директан, јер је решавање Q истовремено и решавање познатог NP-комплетног проблема. Посматрајмо, на пример, проблем **покривање скупова** покривање скупова. Улаз је колекција подскупова $S_1, S_2, \ldots, S_n$ задатог скупа U и природни број k. Проблем је установити да ли постоји подскуп $W \subseteq U$ са највише k елемената, који садржи бар по један елеменат сваког од подскупова S_i. Запажамо да је проблем покривач грана специјални случај проблема покривања скупова у коме U одговара скупу чворова V, а сваки скуп S_i одговара једној грани и садржи два чвора којима је та грана суседна. Према томе, ако знамо да решимо проблем покривања скупова за произвољне скупове, онда можемо да решимо и проблем покривач грана.

Морамо, међутим, бити пажљиви кад користимо овај приступ. У општем случају није тачно да додавање нових захтева чини проблем тежим. Посматрајмо проблем покривач грана. Претпоставимо да смо додали ограничење да покривач грана не сме да садржи два суседна чвора. Другим речима, тражимо мали скуп чворова који је истовремено и покривач грана и независан скуп (независан скуп је подскуп чворова графа, такав да између његова два произвољна елемента не постоји грана). Овај проблем је на први поглед тежи и од проблема покривач грана и од проблема независан скуп, јер треба бринути о више захтева. Испоставља се, међутим, да је ово лакши проблем, и да се може решити за полиномијално време. Разлог овој појави је у томе што допунски захтеви толико сужавају фамилију подскупова кандидата, да се минимум лако налази.

Друга релативно једноставна техника користи **локалне редукције**. Објекат, улаз за један проблем, пресликава се у објекат који је улаз за други проблем. Пресликавање се изводи на локални начин, независно од осталих објеката. Доказ NP-комплетности проблема доминирајућег скупа изведен је на овај начин. Свака

грана у једном графу замењена је троуглом у другом графу. Потешкоћа са овом техником је како на најбољи начин дефинисати одговарајуће објекте.

Најкомпликованија техника је употреба саставних елемената – блокова, као што је то учињено при доказу NP-комплетности проблема 3-обојивост. Блокови обично зависе један од другог, па је њихово независно пројектовање неизводљиво. Морају се имати на уму сви циљеви проблема, да би се могло координирати пројектовање различитих блокова.

7.4.8 Још неки NP-комплетни проблеми

Следи списак садржи још неколико NP-комплетних проблема, који могу бити корисни као полазна основа за нове редукције. Проналажење правог проблема за редукцију обично је више од половине доказа NP-комплетности.

Хамилтонов циклус: Хамилтонов циклус у графу је прост циклус који садржи сваки чвор графа тачно једном. Проблем је установити да ли задати граф садржи Хамилтонов циклус. Проблем је NP-комплетан и за неусмерене и за усмерене графове. (Редукција проблема покривач грана.)

Хамилтонов пут: Хамилтонов пут у графу је прости пут који садржи сваки чвор графа тачно једном. Проблем је установити да ли задати граф садржи Хамилтонов пут. Проблем је NP-комплетан и за неусмерене и усмерене графове. (Редукција проблема покривач грана.)

Проблем трговачког путника: Нека је задат тежински комплетан граф G и број W. Установити да ли у G постоји Хамилтонов циклус са збиром тежина грана $\leq W$. (Директна редукција проблема Хамилтонов циклус.)

Независан скуп: Независан скуп у графу је подскуп чворова графа, такав да између његова два произвољна елемента не постоји грана. Ако је задат граф G и природни број k, установити да ли G садржи независни скуп са бар k чворова. (Директна редукција проблема клика.)

3-димензионално упаривање: Нека су X, Y и Z дисјунктни скупови од по k елемената, и нека је M задати скуп тројки (x, y, z) таквих да је $x \in X$, $y \in Y$ и $z \in Z$. Проблем је установити да ли постоји такав подскуп скупа M који сваки елеменат садржи тачно једном. Одговарајући дводимензионални проблем упаривања је обичан проблем бипартитног упаривања. (Редукција проблема 3 SAT.)

Партиција: Улаз је скуп X чијем је сваком елементу x придружена његова величина $s(x)$. Проблем је установити да ли је могуће поделити скуп на два дисјунктна подскупа са једнаким сумама величина. (Редукција проблема 3-димензионално упаривање.)

Приметимо да се овај и следећи проблем могу ефикасно решити алгоритмом заснованим на динамичком програмирању ако су величине мали цели бројеви. Међутим, пошто је величина улаза сразмерна броју бита потребних да се представи улаз, овакви алгоритми (који се зову **псеудополиномијални алгоритми**) су уствари експоненцијални у односу на величину улаза.

Проблем ранца: Улаз су два броја S, V и скуп X чијем је сваком елементу x придружена величина $s(x)$ и вредност $v(x)$. Проблем је установити да ли постоји подскуп $B \subseteq X$ са укупном величином $\leq S$ и укупном вредношћу $\geq V$. (Редукција проблема партиције.)

Проблем паковања: Улаз је низ бројева $a_1, a_2, \ldots, a_n$ и два броја b, k. Проблем је установити да ли се овај скуп може разложити у k подскупова, тако да је сума бројева у сваком подскупу $\leq b$. (Редукција проблема партиције.)

7.5 Технике за рад са NP-комплетним проблемима

Појам NP-комплетности је основа за елегантну теорију која омогућује препознавање проблема за које највероватније не постоји полиномијални алгоритам. Међутим, доказивањем да је проблем NP-комплетан, сам проблем није елиминисан! И даље је потребно решити га. Технике за решавање NP-комплетних проблема су понекад другачије од техника које смо до сада разматрали. Ни један NP-комплетан проблем се (највероватније) не може решити тачно и комплетно алгоритмом полиномијалне временске сложености. Због тога смо принуђени на компромисе. Најчешћи компромиси односе се на оптималност, гарантовану ефикасност, или комплетност решења. Постоје и друге алтернативе, од којих свака понешто жртвује. Исти алгоритам се може користити у различитим ситуацијама, примењујући различите компромисе.

Алгоритам који не даје увек оптималан (или тачан) резултату зове се **приближан алгоритам**. Посебно су занимљиви приближни алгоритми који могу да гарантују границу за степен одступања од тачног решења. Нешто касније видећемо три примера таквих алгоритама.

У одељку 3.1 разматрали смо пробабилистичке алгоритме који могу да погреше. Најпознатији у тој класи је алгоритам за препознавање простих бројева. За проблем препознавања простих бројева се до 2002. године није знало да ли је у P, док Агравал, Кајал и Саксена пронашли алгоритам полиномијалне сложености (Agrawal–Kayal–Saxena primality test; врло неефикасан алгоритам, неупоредиво неефиксанији од пробабилистичког!) који решава овај проблем. Овде се нећемо бавити алгоритмима за препознавање простих бројева, јер захтевају предзнање из теорије бројева. Раширено је веровање да се NP-комплетни проблеми не могу решити алгоритмом полиномијалне временске сложености, код којих је вероватноћа грешке мала за све улазе. Такви алгоритми су по свему судећи ефикасни за проблеме, за које се не зна да ли су у класи P, али се не верује да су NP-комплетни. Такви проблеми нису чести. Пробабилистички алгоритми се могу користити као део других стратегија — на пример, као део приближних алгоритама.

Други компромис могућ је у вези са захтевом да полиномијално буде време извршавања у *најгорем случају*. Може се покушати са решавањем NP-комплетних проблема за полиномијално *средње време*. Потешкоћа са овим приступом је како дефинисати *просечно време*. На пример, тешко је искључити улазе за које је конкретан проблем тривијалан (као што је граф који има само изоловане чворове) из израчунавања просека. Такви тривијални улази могу знатно да утичу на просек. Алгоритми предвиђени за поједине типове случајних улаза могу да буду корисни ако је стварна расподела вероватноћа улаза у складу са претпостављеном. Међутим, обично је налажење тачне расподеле је обично врло тешко. Најтежи део посла при конструкцији алгоритама, који у просеку добро раде, је најчешће њихова анализа.

Коначно, могу се правити компромиси у вези са комплетношћу алгоритама; наиме, може се дозволити да алгоритам ради ефикасно само за неке специјалне улазе. На пример, проблем покривач грана може се решити за полиномијално време за бипартитне графове. Према томе, кад се формулише апстрактни проблем полазећи од ситуације из реалног живота, треба обезбедити да сви допунски услови које улаз задовољава буду укључени у апстрактну дефиницију. Други пример су алгоритми са експоненцијалном временском сложеношћу, који се ипак могу извршавати за мале улазе, што је често потпуно задовољавајуће.

У овом одељку описаћемо више оваквих техника и илустровати их примерима. Започињемо са две опште и корисне технике које се зову **претрага** и **гранање са одсецањем**. Ове технике су сличне. Могу се користити као основа за приближни алгоритам, или као тачан алгоритам за мале улазе. На крају наводимо неколико примера приближних алгоритама.

Илустроваћемо ову идеју проблемом целобројног линеарног програмирања, споменутог у одељку 6.3. Проблем је сличан линеарном програмирању, али има допунско ограничење да промењиве могу узимати само целобројне вредности. Нека је $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ вектор колона промењивих, нека су $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_k$ вектори врсте реалних бројева једнаке дужине n, и нека су $b_1, b_2, \ldots, b_k, c_1, c_2, \ldots, c_n$ реални бројеви. Проблем је максимизирати вредност **линеарне циљне функције**

$$z = \sum_{i=1}^{n} c_i x_i$$

под условом да су координате вектора $\mathbf{x}$ цели бројеви и

$$\mathbf{a}_j \cdot \mathbf{x} \leq b_j, \quad j = 1, 2, \ldots, k,$$

при чему су сви симболи сем $\mathbf{x}$ константе. Многи NP-комплетни проблеми могу се лако формулисати као проблеми целобројног линеарног програмирања (у наставку ћемо видети један такав пример). Према томе, целобројно линеарно програмирање је NP-тежак проблем. Прецизније, проблем је NP-комплетан, али је доказ да је тај проблем у класи NP нешто компликованији.

Показаћемо сада како се проблем клика може формулисати као проблем целобројног линеарног програмирања При томе разматрамо варијанту у којој се тражи максимална клика, уместо да се проверава да ли постоји клика задате величине. Уводимо n промењивих $x_1, x_2, \ldots, x_n$, које одговарају чворовима, тако да је $x_i = 1$ ако v_i припада максималној клики, односно $x_i = 0$ у противном. Циљна функција је

$$z = x_1 + x_2 + \cdots + x_n,$$

јер је потребно укључити у клику што већи број чворова. Постоји по једно ограничење за сваки чвор

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

и по једно ограничење за сваки пар несуседних чворова

$$x_i + x_j \leq 1 \text{ за сваки пар чворова } v_i, v_j \text{ таквих да } (v_i, v_j) \notin E.$$

Први скуп ограничења ограничава вредности промењивих на 0 или 1. Други скуп ограничења обезбеђује да се два несуседна чвора не могу истовремено изабрати у клику; према томе, чворови који су изабрани чине клику.

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 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 промењиве 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 параметара и напредујемо дуж стабла, при чему теме ниже у стаблу одговара потпроблему 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 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. Учестаност одсецања, односно еифкасност целог алгоритма, зависи од хеуристике за формирање потпроблема и избора наредног потпроблема за испитивање. Хеуристика зависи од конкретног проблема, и у овој области спроводе се широка истраживања.

7.5.1 Приближни алгоритми са гарантованим квалитетом решења

У овом одељку размотрићемо приближне алгоритме за три NP-комплетна алгоритма: покривач грана, паковање и еуклидски проблем трговачког путника. Сва три алгоритма имају гарантовани квалитет решења; другим речима, може се доказати да решења која они дају нису предалеко од оптималних решења.

Покривач грана

Најпре ћемо размотрити једноставан приближни алгоритам за налажење покривача грана датог графа. Алгоритам гарантује налажење покривача са највише два пута више чворова него у минималном покривачу. Нека је G граф, и нека је M максимално упаривање у G. Пошто је M упаривање, његове гране немају заједничких тачака, а пошто је M максимално упаривање, све остале гране имају бар један заједнички чвор са неком граном из M.

Теорема 17. *Скуп чворова суседних гранама максималног упаривања M је покривач грана, са највише два пута више чворова него што их има минимални покривач.*

Доказ. Скуп чворова који припадају M чини покривач грана, јер је M максимално упаривање. Сваки покривач грана мора да покрије све гране — специјално, све гране упаривања M. Пошто је M упаривање, било који чвор из M може да покрије највише једну грану упаривања M. Према томе, бар половина чворова из M мора да припада покривачу грана.

Максимално упаривање може се наћи простим додавањем грана све дотле док је то могуће (односно док постоје нека два неупарена, а суседна чвора).

Једнодимензионално паковање

Једнодимензионални проблем паковања односи се на паковање објеката различите величине у *кутије* тако да се искористи најмањи могући број кутија. На пример, приликом селидбе је циљ пренети све ствари, користећи камион најмањи могући број пута, пакујући ствари што је могуће боље. То је наравно тродимензионални проблем; овде ћемо се позабавити његовом једнодимензионалном верзијом. Због једноставности се претпоставља да сви сандуци имају величину 1.

Проблем. Нека је $x_1, x_2, \ldots, x_n$ скуп реалних бројева између 0 и 1. Поделити их у најмањи могући број подскупова, тако да сума бројева у сваком подскупу буде највише 1.

На једнодимензионални проблем паковања наилази се, на пример, у проблемима управљања меморијом, кад постоје захтеви за доделом меморијских блокова различите величине, а додељивање се врши из неколико једнаких великих блокова меморије. Једнодимензионални проблем паковања је NP-комплетан.

Једна од могућих хеуристика за решавање овог проблема је ставити x_1 у прву кутију, а затим, за свако i, ставити x_i у прву кутију у којој има довољно места, или започети са новом кутијом, ако нема довољно места ни у једној од коришћених кутија. Овај алгоритам зове се **први одговарајући** и, као што показује следећа теорема, довољно је добар у најгорем случају.

Теорема 18. *Алгоритам први одговарајући захтева највише $2m$ кутија, где је m најмањи могући број кутија.*

Доказ. По завршетку алгоритма први одговарајући не постоје две кутије са искоришћењем мањим од 1/2. Према томе, ако са k означимо број употребљених кутија, биће $m \geq \sum_{i=1}^{n} x_i > (k-1)/2$, одакле је $k < 2m+1$, односно $k \leq 2m$.

Испоставља се да је граница дефинисана овом теоремом прилично груба. Константа 2 из теореме може се смањити на 1.7 после нешто компликованије анализе. Константа 1.7 не може се даље смањити, јер постоје примери у којима алгоритам први одговарајући захтева 1.7 пута више кутија од оптималног алгоритма. Описани алгоритам може се једноставно побољшати. До најгорег случаја долази кад се много малих бројева појављује на почетку. Уместо да се бројеви пакују у редом којим наилазе, они се најпре сортирају у нерастући низ. Промењени алгоритам зове се **опадајући први одговарајући**, и у најгорем случају троши највише око 1.22 пута више кутија од оптималног алгоритма (теорему дајемо без доказа).

Теорема 19. *Алгоритам опадајући први одговарајући захтева највише $\frac{11}{9}m+4$ кутија, где је m најмањи могући број кутија.*

Константа 11/9 је такође најбоља могућа. Први одговарајући и опадајући први одговарајући су једноставне хеуристике. Постоје други методи који гарантују још мање константе. Њихова анализа је у већини случајева компликована.

Стратегије које смо описали типичне су за хеуристичке алгоритме. Оне одражавају природне приступе, односно одговарају начину на који би неко ручно решавао ове проблеме. Међутим, видели смо много случајева у којима се директни приступи понашају лоше за велике улазе. Због тога је веома важно анализирати понашање таквих алгоритама.

Еуклидски проблем трговачког путника

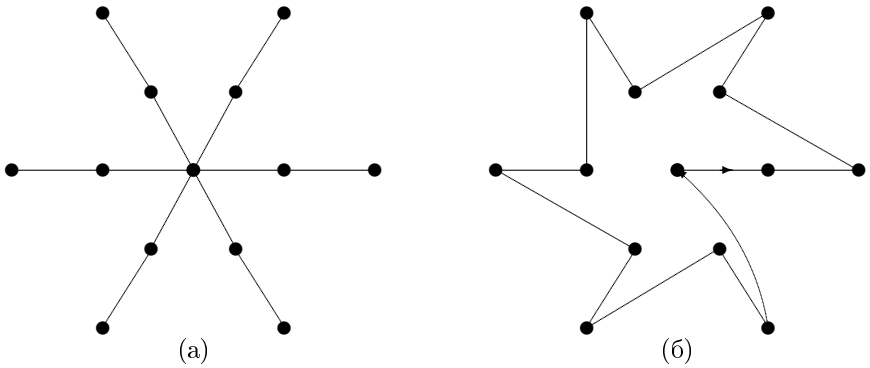
Проблем трговачког путника (TSP, скраћеница од traveling salesman problem) је важан проблем са много примена. Размотрићемо овде варијанту TSP са допунским ограничењем да тежине грана одговарају еуклидским растојањима.

Проблем. Нека је $C_1, C_2, \ldots, C_n$ скуп тачака у равни које одговарају положајима n градова. Пронаћи Хамилтонов циклус минималне дужине (маршруту трговачког путника) међу њима.

Проблем је и даље NP-комплетан (ово тврђење наводимо без доказа), али видећемо да претпоставка да су растојања еуклидска омогућује конструкцију приближног алгоритма за његово решавање. Прецизније, ова претпоставка може се уопштити, замењујући је претпоставком да растојања задовољавају **неједнакост троугла**, односно да је растојање између два чвора увек мање или једнако од дужине произвољног пута између њих преко осталих чворова.

Најпре се конструише минимално повезујуће стабло (MCST; овде су цене грана њихове дужине) познатим алгоритмом полиномијалне сложености. Тврдимо да цена стабла није већа од дужине најбољег циклуса TSP. Заиста, циклус TSP садржи све чворове, па га уклањање једне гране чини повезујућим стаблом, чија цена је већа или једнака од цене MCST.

Од повезујућег стабла се, међутим, не добија директно циклус TSP. Посматрајмо најпре циклус који се добија претрагом у дубину овог стабла (полазећи од произвољног чвора), и садржи грану у обрнутом смеру увек кад се грана приликом враћања пролази у супротном смеру (овај циклус одговара, на пример, обиласку галерије у облику стабла, са сликама на оба зида сваког ходника, идући увек удесно). Свака грана се тако пролази тачно два пута, па је цена овог циклуса највише два пута већа од цене MCST. На крају се овај циклус преправља у циклус TSP, идући пречицом увек кад треба проћи кроз грану већ укључену у циклус (видети слику 7.9). Другим речима, уместо повратка старом граном, идемо директно до првог непрегледаног чвора. Претпоставка да су растојања еуклидска је важна, јер обезбеђује да је увек директни пут између два града бар толико добар, колико било који заобилазни пут. Према томе, дужина добијеног циклуса TSP је још увек мања од двоструке дужине минималног циклуса TSP.

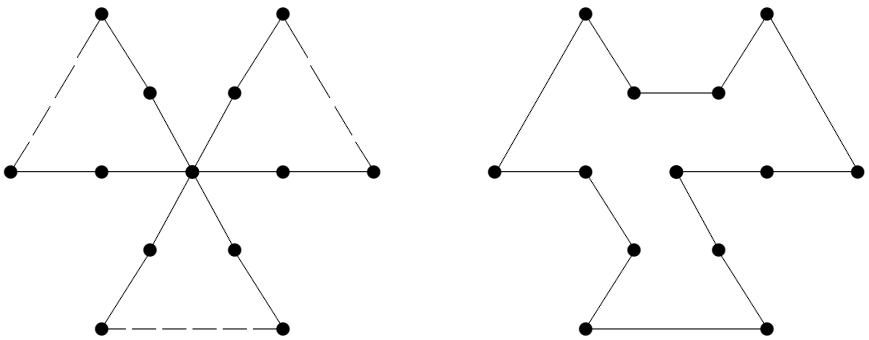


Слика 7.9: (а) Минимално повезујуће стабло (MCST) (б) циклус TSP добијен од MCST полазећи од средњег чвора и идући увек удесно.

Сложеност. Временска сложеност овог алгоритма одређена је бројем корака у алгоритму за конструкцију MCST.

Алгоритам који смо управо описали може се побољшати на следећи начин. Његов "најгрубљи" део је претварање обиласка стабла у циклус TSP. Та конверзија може се посматрати и на други начин: она формира Ојлеров циклус од стабла у коме је свака грана удвостручена. После тога се конструише циклус TSP коришћењем пречица у Ојлеровом циклусу. Конверзија стабла у Ојлеров циклус може се извести много рационалније. Ојлеров граф може да садржи само чворове парног степена. Посматрајмо све чворове непарног стапена у стаблу. Број таквих чворова мора бити паран (у противном би сума степена чворова била непарна, што је немогуће, јер је та сума једнака двоструком броју грана). Ојлеров граф се од стабла може добити додавањем довољног броја грана, чиме се постиже да степени свих чворова постану парни. Пошто се циклус TSP састоји од Ојлеровог циклуса (са неким пречицама), волели бисмо да минимизирамо збир дужина додатих грана. Размотримо сада тај проблем.

Дато је стабло у равни, а циљ је додати му неке гране, са минималном сумом дужина, тако да добијени граф буде Ојлеров. Сваком чвору непарног степена мора се додати бар једна грана. Покушајмо да постигнемо циљ додавањем тачно једне гране сваком таквом чвору. Претпоставимо да има $2k$ чворова непарног степена. Ако додамо k грана, тако да свака од њих спаја два чвора непарног степена, онда ће степени свих чворова постати парни. Проблем тако постаје проблем *упаривања*. Потребно је пронаћи упаривање минималне дужине које покрива све чворове непарног степена. Налажење оптималног упаривања може се извести за $O(n^3)$ корака за произвољни граф (ово тврђење наводимо без доказа). Коначни циклус TSP се затим добија од Ојлеровог графа (који обухвата минимално повезујуће стабло и упаривање минималне дужине) коришћењем пречица. Циклус TSP добијен овим алгоритмом од стабла са слике 7.9 приказан је на слици 7.10.



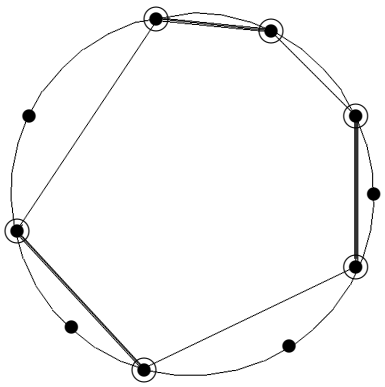
Слика 7.10: Минимални Ојлеров циклус и одговарајући циклус TSP (а) минимално повезујуће стабло са упаривањем, (б) циклус TSP добијен од Ојлеровог циклуса.

Теорема 20. *Побољшани алгоритам даје циклус* TSP *чија је дужина највише* 1.5 *пута већа од дужине минималног циклуса* TSP.

ф

Доказ. Потребно је оценити дужину Ојлеровог циклуса, јер се дужина циклуса после увођења евентуалних пречица може само смањити. Ојлеров циклус се састоји од стабла и упаривања. Означимо са Q минимални циклус TSP, а са $|Q|$ његову дужину. Видели смо већ да је дужина стабла мања или једнака од $|Q|$;

према томе, довољно је доказати да дужина упаривања не прелази $|Q|/2$. Циклус Q садржи све чворове. Нека је D скуп чворова непарног степена у полазном стаблу. За скуп D могу се формирати два дисјунктна упаривања са збиром дужина $\leq |Q|$ на следећи начин (видети слику 7.11, на којој су заокружени чворови из D). Започињемо са произвољним чвором $v \in D$ и упарујемо га са са најближим чвором (у смеру казаљке на сату) дуж циклуса Q. После тога наставља се са упаривањем у истом смеру. Ако упарени чворови нису суседи у Q, онда је растојање између њих мање или једнако од дужине пута који их повезује у Q (због неједнакости троугла). Овај процес даје једно упаривање. Друго упаривање добија се понављањем истог процеса у смеру супротном од казаљке на сату. Сума дужина оба упаривања је највише $|Q|$, видети слику 7.11. Међутим, пошто је M упаривање минималне тежине у D, његова дужина је мања или једнака од дужине бољег од два споменута упаривања (са збиром $|Q|$), па је дакле мања или једнака од $|Q|/2$.



Слика 7.11: Два упаривања чија сума дужина не прелази дужину минималног циклуса TSP.

Налажење упаривања минималне тежине знатно је сложеније од налажења минималног повезујућег стабла, али се тиме добија боља граница. Наведени пример илуструје основну карактеристику овог типа алгоритама: уопштава се лакши проблем — или се ослабљују (релаксирају) неки елементи полазног проблема — и тако се формира хеуристика.

Доказ непостојања приближног алгоритма за решавање (општег) проблема трговачког путника

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 7.2.

Теорема 7.2 (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

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

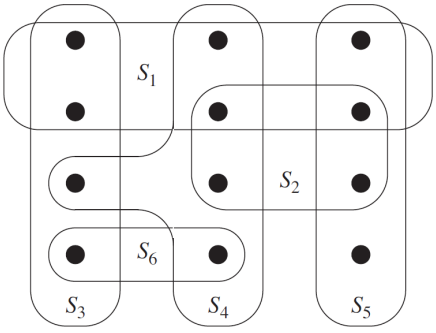
Дакле, цена Хамилтоновог циклуса у G' је барем за фактор $\rho + 1$ већа од цене Хамилтоновог циклуса у G' који јесте Хамилтонов циклус у G.

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.

7.5.2 Приближни алгоритам за тражење минималног покривача скупа

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.



Слика 7.12: 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.

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 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:

$$X = \bigcup_{S \in \mathcal{C}} S. \tag{7.1}$$

Ако нека фамилија $\mathcal{C}$ задовољава једнакост 7.1, онда кажемо да она *покрива* скуп X. На слици 7.12 приказан је пример проблема минималног покривача скупа; минимални покривач скупа је $\mathcal{C} = \{S_3, S_4, S_5\}$, 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, видети код на слици 7.13.

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
$i \leftarrow 0$ // редни број итерације
$U \leftarrow X$
$\mathcal{C} \leftarrow \emptyset$
while $U \neq \emptyset$ **do**
Одабери скуп $S \in \mathcal{F}$ који максимизује вредност $|S \cap U|$
$i \leftarrow i + 1$
$U \leftarrow U \setminus S$
$\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
// нека су S_i, U_i, $\mathcal{C}_i$ вредности S, U, $\mathcal{C}$ у овом тренутку
end

Слика 7.13: Похлепни алгоритам за покривање скупа.

У примеру на слици 7.12, 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 да 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. У доказу се користи једноставна неједнакост .

Лема 7.2. За свако $k > 0$ важи неједнакост $\left(1 - \frac{1}{k}\right)^k < e^{-1}$.

Доказ. Извод $f'(x) = e^x - 1$ функције $f(x) = e^x - x - 1$ је негативан за $x < 0$, односно позитиван за $x > 0$. Због тога у тачки $x = 0$ функција $f(x)$ има минимум 0, тј. за свако x је $e^x \geq 1 + x$. Заменом $x = -1/k$, ова неједнакост постаје $e^{-1/k} \geq 1 - 1/k$. Степеновањем ове неједнакости на k добија се тражена неједнакост.

Теорема 7.3 (Фактор апроксимације похлепног алгоритма за решавање SCP)**.** *Приказани похлепни алгоритам је приближни алгоритам полиномијалне временске сложености са фактором апроксимације*

$$\rho(n) = \lceil \ln |X| \rceil.$$

Доказ. Већ смо показали да је временска сложеност приказаног алгоритма полиномијална.

Нека је $k = |\mathcal{C}^*|$ величина оптималног покривача $\mathcal{C}^*$. Оптимални покривач $\mathcal{C}^*$ величине k покрива скуп $|X|$, па покрива и скуп $U_i \subseteq X$ за свако $i \geq 1$. Због тога постоји скуп $S' \in \mathcal{C}^*$ који покрива бар $\max\{1, |U_{i-1}|/k\}$ елемената скупа U_{i-1}

- због $U_i \neq \emptyset$ постоји скуп $S' \in \mathcal{C}^*$ који покрива бар 1 елемент подскупа U_i;
- у противном (ако ни један подскуп из $\mathcal{C}^*$ не покрива бар $|U_{i-1}|/k$ елемената скупа $|U_{i-1}|$), сви скупови из $\mathcal{C}^*$ покривају строго мање од $k(|U_i|/k) = |U_i|$ елемената скупа U_i, тј. не покривају скуп U_i.

Из $|S'| \geq 1$ следи да S' није у фамилији $\mathcal{C}_{i-1}$, па и скуп $S_i \in \mathcal{F}$ изабран у првом кораку **while** петље покрива бар $\max\{1, |U_{i-1}|/k\}$ елемената скупа U_{i-1}, тј. важи неједнакост $|S \cap U_{i-1}| \geq |S' \cap U_{i-1}| \geq |U_{i-1}|/k$.

Према томе, за сваку итерацију $i \geq 0$ важи неједнакост

$$|U_i| \leq |U_{i-1}| - |U_{i-1}|/k = |U_{i-1}| \left(1 - \frac{1}{k}\right).$$

Полазећи од $|U_0| = |X|$, итерирањем ове неједнакости добија се

$$|U_i| \leq |U_0| \left(1 - \frac{1}{k}\right)^i = |X| \left(1 - \frac{1}{k}\right)^i.$$

Из **while** петље алгоритма се сигурно излази ако је $|X| \left(1 - \frac{1}{k}\right)^i < 1$, јер је тада $|U_i| = 0$. Укупан број итерација i алгоритма можемо да представимо у облику $i = ck$, где је c константа коју треба да одредимо. Узимајући у обзир претходно доказану неједнакост $\left(1 - \frac{1}{k}\right)^k < e^{-1}$, добија се

$$|X| \left(1 - \frac{1}{k}\right)^i = |X| \left(1 - \frac{1}{k}\right)^{ck} < |X| e^{-c}.$$

Ако се за константу c узме целобројна вредност $\lceil \ln |X| \rceil \geq \ln |X|$, тада је

$$|U_i| \leq |X| \left(1 - \frac{1}{k}\right)^i < |X| e^{-c} < 1.$$

Другим речима, добија се да је

$$\mathcal{C} \leq i = ck = c|\mathcal{C}^*| = \lceil \ln |X| \rceil |\mathcal{C}^*|,$$

што је требало доказати.

7.6 Резиме

Претходна поглавља су можда изазвале оптимизам у вези са могућношћу конструкције добрих алгоритама. Ово поглавље треба да нас приближи реалности. Има много важних проблема који се, нажалост, не могу решити елегантним, ефикасним алгоритмима. Потребно је да препознајемо такве проблеме и да им проналазимо неоптимална, приближна решења. При нападу на неки проблем на располагању су два приступа. Може се покушати са техникама из претходних поглавља, да би се проблем решио, или могу искористити технике уведене у овом поглављу, и показати да је проблем NP-комплетан. Да би се избегли многобројни безуспешни покушаји пре избора исправног приступа, потребно је развити интуицију за оцену тежине проблема.

Глава 8

Паралелни алгоритми

8.1 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.

8.2 Модели паралелног израчунавања

Детаљан преглед модела паралелних рачунара захтевао би више простора. Споменућемо 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 8.4), 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 8.3.

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 8.4.

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 8.5.

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.

8.3 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 CRCW (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 8.3.2 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.

8.3.1 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.

8.3.2 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 пронађи 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 8.1 (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.

8.3.3 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 8.1. Чињеница да се проласци кроз **do** петљу извршавају паралелно (истовремено, на различитим скуповима процесора) у коду је назначена додатком "in parallel".

```
Алгоритам Paralelni_Prefiks_1(x, n);
Улаз: x (низ са n елемената).
{претпоставља се да је n степен двојке}
Излаз: x (чији i-ти елемент 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] ⋆ x[Desni] {⋆ 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] ⋆ x[i]
end
```

Слика 8.1: Алгоритам *Paralelni_prefiks_1*.

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 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 8.2.

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 извршава паралелно.

8.3.4 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.

Алгоритам Paralelni_Prefiks_2(x, n);
Улаз: x (низ са n елемената).
{претпоставља се да је n степен двојке}
Излаз: x (чији i-ти елемент садржи i-ти префикс).
begin
    $PP\_2(1)$
end
procedure PP_2$(Korak)$;
begin
    if $Korak = n/2$ **then**
        $x[n] := x[n/2] \star x[n]$ {$\star$ је асоцијативна бинарна операција}
    else
        for $i := 1$ **to** $n/(2 \cdot Korak)$ **do in parallel**
            $x[2 \cdot i \cdot Korak] := x[(2 \cdot i - 1) \cdot Korak] \star x[2 \cdot i \cdot Korak]$;
        $PP\_2(2 \cdot Korak)$;
        for $i := 1$ **to** $n/(2 \cdot Korak) - 1$ **do in parallel**
            $x[(2 \cdot i + 1) \cdot Korak] := x[2 \cdot i \cdot Korak] \star x[(2 \cdot i + 1) \cdot Korak]$
end

Слика 8.2: Алгоритам $Paralelni\_prefiks\_2$.

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 8.3) 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 8.3: 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 $O(n \log n)$.

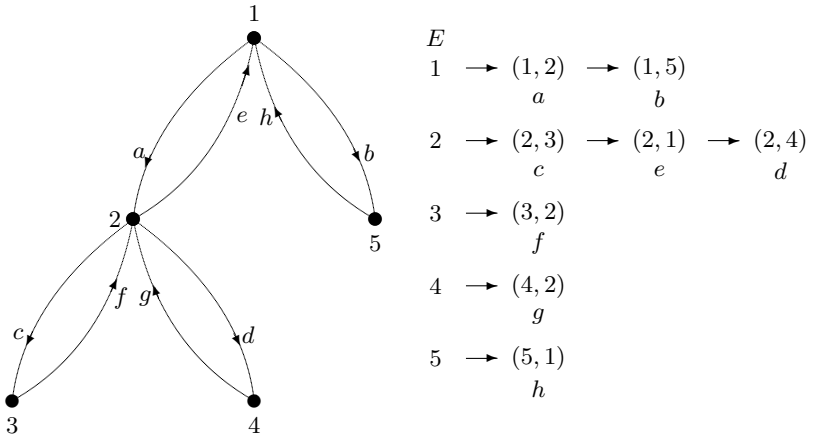
Познавање рангова омогућује трансформацију листе у низ за $O(\log n)$ паралелних корака. После израчунавања свих рангова, елементи се могу паралелно прекопирати на своје локације у низу, па се остатак израчунавања може извршити директно на низу, после чега је њихова обрада много једноставнија.

8.3.5 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 8.4; показивачи на копије грана због прегледности нису приказани.



Слика 8.4: 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 8.4, 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 8.3 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 $T(n, n) = O(\log n)$

8.4 Алгоритми за мреже рачунара

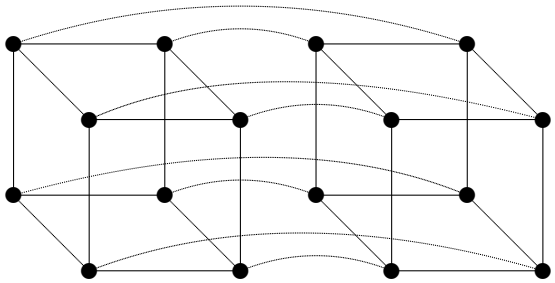
Мреже рачунара могу се моделирати графовима, обично неусмереним. Процесори одговарају чворовима, а два чвора су повезана граном ако постоји директна веза између одговарајућих процесора. Сваки процесор има своју локалну меморију, а посредством мреже може да приступи локалним меморијама других процесора. 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 8.5 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 умеђући мреже уместо темена хиперкоцке. У примени се појављују и друге структуре мрежа.

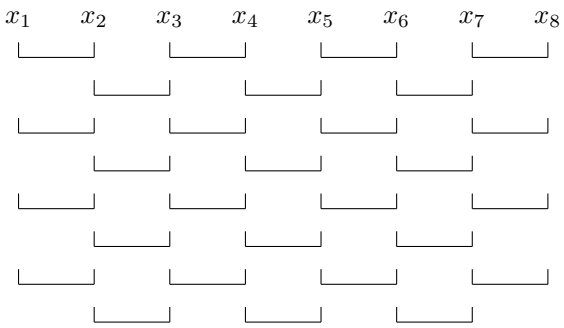


Слика 8.5: Четвородимензионална хиперкоцка.

8.4.1 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 8.6). 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 8.7. Пример рада алгоритма приказан је на слици 8.8. 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 8.6: Сортирање парно-непарним транспозицијама.

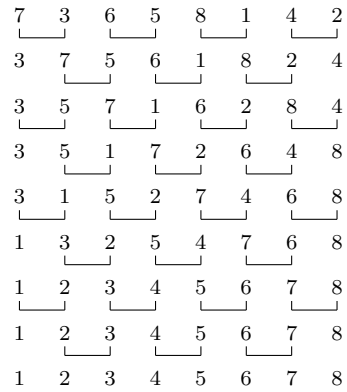
Алгоритам Sortiranje_na_nizu(x, n);
Улаз: x (низ са n елемената, при чему је x_i у процесору P_i).
Излаз: x (сортирани низ, тако да је i-ти најмањи елеменат у P_i).
begin
  do in parallel $\lceil n/2 \rceil$ пута
    P_{2i-1} и P_{2i} упоређују своје елементе и по потреби их размењују;
      {за све 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 8.7: Алгоритам за сортирање на низу процесора.

Алгоритам *Sortiranje_na_nizu* изгледа природно и јасно, али доказ исправности његовог рада није тривијалан. На пример, елеменат се у неким корацима може *удаљавати* од свог коначног положаја. У примеру на слици 8.8 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 21. *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

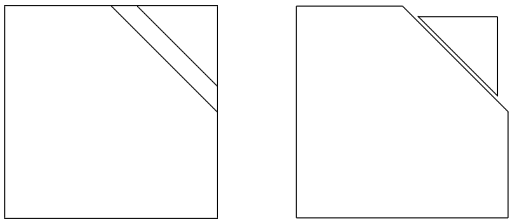


Слика 8.8: Пример сортирања парно-непарним транспозицијама.

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 елемент, и претпоставимо да је то x_m (у примеру на слици 8.8 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 8.8), 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 8.9). 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 8.8. 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 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.



Слика 8.9: Корак индукције у доказу исправности сортирања парно-непарним транспозицијама, Теорема 21.

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.

8.4.2 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.

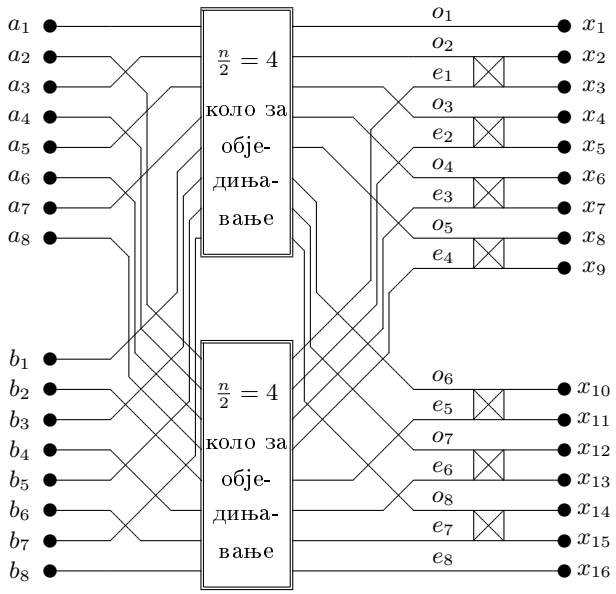
Описаћемо сада другачији алгоритам обједињавања, заснован на техници разлагања. Претпоставимо због једноставности да је n степен двојке. Нека су $a_1, a_2, \ldots, a_n$; $b_1, b_2, \ldots, b_n$ два сортирана низа које треба објединити, и нека је $x_1, x_2, \ldots, x_{2n}$ резултат њиховог обједињавања. 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 $a_1, a_3, \ldots, a_{n-1}$; $b_1, b_3, \ldots, b_{n-1}$, 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 22. *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

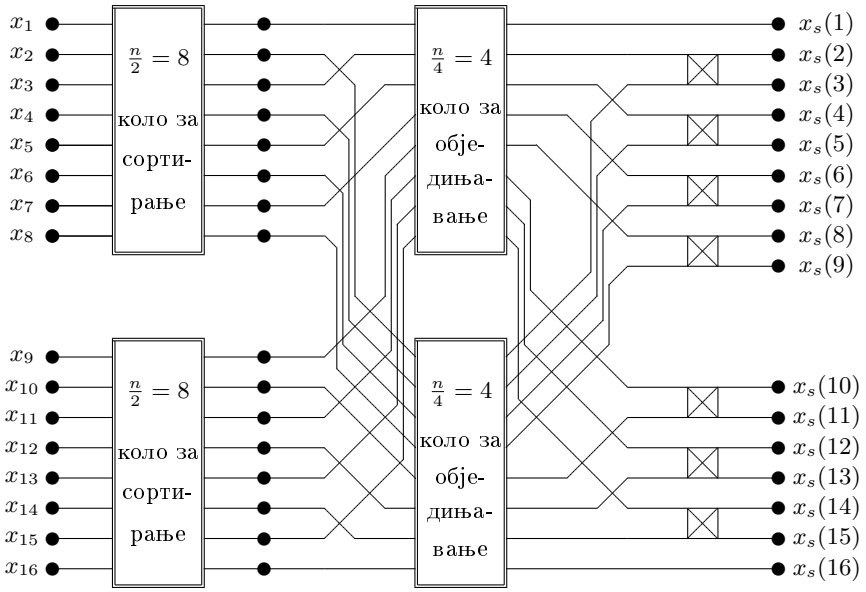
или једнако од *бар* i парних елемената у оба низа. Међутим, сваком парном елементу који је мањи или једнак од e_i одговара још један непарни елемената који је такође мањи или једнак од e_i (јер смо пошли од два сортирана низа). Према томе, e_i је веће или једнако од бар $2i$ елемената из оба низа. Другим 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 22 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 8.10 илуструје рекурзивну конструкцију обједињавања, а на слици 8.11 се види пример комплетног сортирања, које се зове **сортирање парно-непарним обједињавањем**. Правоугаоници на левој страни слике 8.11, означени са "$n/2$ сортирање", представљају рекурзивне копије комплетне мреже за сортирање, које сортирају по $n/2$ бројева.



Слика 8.10: Коло за парно–непарно обједињавање два низа од по $n = 8$ елемената.

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",



Слика 8.11: Пример сортирања 16 бројева помоћу кола за сортирање парно-непарним обједињавањем.

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.

8.4.3 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.

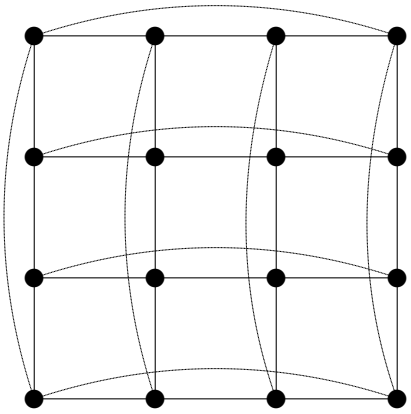
После прве итерације појављује се један проблем; пошто су неки елементи елиминисани, такмичари више нису равноправни. Посматрајмо, на пример, екстремни случај кад су у једној половини стабла елиминисани сви листови сем једног. Преостали елеменат у тој половини стабла "провући" ће се до "финала" без икаквог 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 $O(\log^2 n)$.

8.4.4 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 8.12. 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).



Слика 8.12: Дводимензионална пресавијена мрежа.

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 $P[0,0]$ 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

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 8.1: Почетно размештање елемената матрица — припрема за паралелно множење.

за i места навише, $i = 0, 1, \ldots, n-1$. Алгоритам је приказан на слици 8.13. На слици 8.1 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.

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 8.13: Алгоритам за паралелно множење матрица на мрежи.

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.

8.5 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.

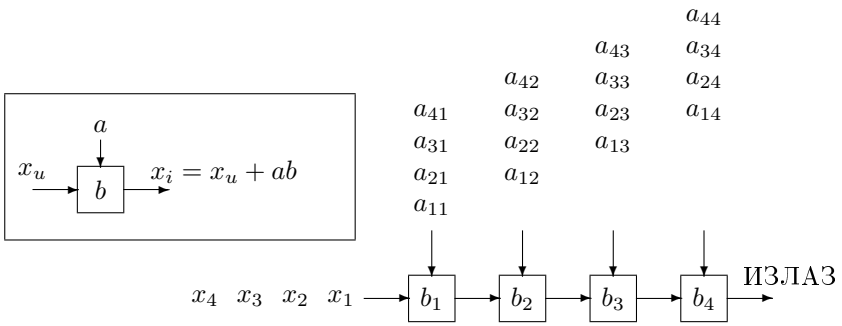
8.5.1 Множење матрице и вектора

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 8.14. 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.

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.



Слика 8.14: Множење вектора матрицом.

8.5.2 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

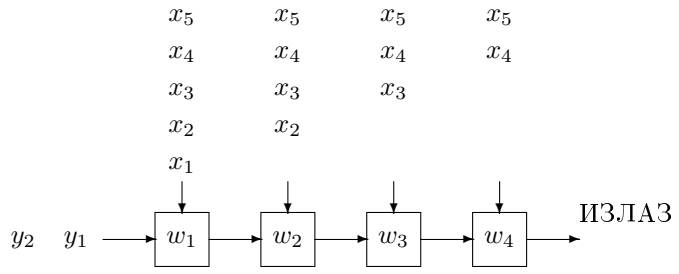
$$y_i = w_1 x_i + w_2 x_{i+1} + \cdots w_k x_{i+k-1}, \quad i = 1, 2, \ldots, n+1-k.$$

Проблем. Израчунати конволуцију y вектора x и w.

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

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

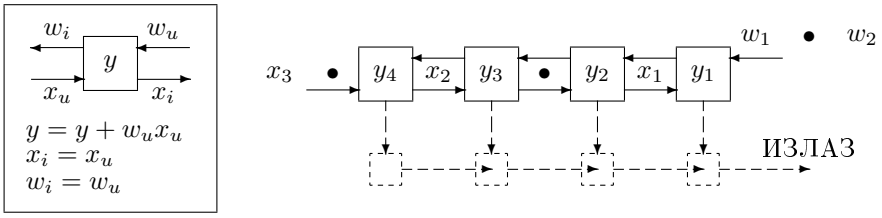
Систолички алгоритам који решава овај проблем може се добити као специјални случај општијег алгоритма за израчунавање производа матрице и вектора (видети претходни проблем, слика 8.14). Ово је приказано на слици 8.15. Приметимо да се x_i истовремено користи дуж целог низа процесора (сем првих $k-1$ вредности 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 8.15: Конволуција са простирањем.

Процесори на слици 8.15 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 8.16 (црне тачке одговарају одсутним подацима).



Слика 8.16: 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 8.16. Недостатак кретања података двоструко мањом брзином је у томе што израчунавање траје двоструко дуже.

8.6 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.

[8] J. Kärkkäinen, P. Sanders, Simple Linear Work Suffix Array Construction, *Lecture Notes in Computer Science* 2719: 943–955.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 4. izdanje, The MIT Press, 2022.

[10] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.

[11] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. U zborniku A. Amir (eds) Combinatorial Pattern Matching CPM 2001. *Lecture Notes in Computer Science* 2089. Springer, 2001.