**УНИВЕРЗИТЕТ У БЕОГРАДУ**

**ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ**

Душан Војиновић, 2017/0080

**ФЕНВИКОВО СТАБЛО**

*дипломски рад*

ментор:

проф. др Милo Томашевић, дипл. инж. ел.

Београд, септембар 2021.

**РЕЗИМЕ**

У овом раду представљено је Фенвиково стабло (*Fenwick Tree*), његова примена на класичном проблему ажурирања и рачунања кумулативних сума (збира елемената низа почевши од првог елемента, па све до елемента са траженим индексом) индекса низа, и поређење са интуитивним решењима истог проблема.

Дата су теоријска разматрања сложености ових операција, имплементација у програмском језику *С++* као и експериментално упоређивање времена извршавања операција коришћењем поменутих техника у зависности од величине низа и учестаности операција ради стицања идеје када је пожељно користити ову технику, а када се држати „класичних“ имплементација.

Након тога, приказане су још неке примене ове структуре података, проширење на дводимензионалне низове и један предлог имплементације дате структуре података користећи конкурентно програмирање.

**Кључне речи:** Фенвик, Фенвиково стабло, кумулативне суме

**САДРЖАЈ**

Садржај .......................... 1

Увод .......................... 2

1. Технике решавања датог проблема .......................... 3

1.1 Наивне технике .......................... 3

1.2 Фенвиково стабло .......................... 5

2. Имплементација Фенвиковог стабла .......................... 13

3. Упоређивање перформанси .......................... 14

4. Примене .......................... 19

4.1 Број инверзија низа .......................... 19

4.2 Рачунање интервала вероватноће догађаја .......................... 19

5. Вишедимензиона Фенвикова стабла .......................... 22

5.1 2D Фенвикова стабла .......................... 22

5.2 D-димензионо (D>2) Фенвиково стабло .......................... 24

6. Паралелизација операција над Фенвиковим стаблом .......................... 26

Коришћени алати и окружење .......................... 30

Закључак .......................... 31

Литература .......................... 32

**УВОД**

Посматрајмо следећи проблем. Нека је дат низ фиксне дужине *N*. Желимо да над овим низом извршавамо следеће операције:

1. Ажурирање: Желимо да дати елемент низа саберемо са одређеном вредности
2. Кумулативна сума: Желимо да за дати индекс дохватимо збир бројева од првог елемента до датог индекса рачунајући и број на датом индексу

Приметимо, такође да се на основу ових операција тривијално могу извршити и следеће операције:

1. Парцијална сума: Сума елемената низа од индекса *l* дo индекса *d,* рачунајући и та два индекса

Може се добити као кумулативна сума индекса *d* умањена за вредност свих елемената низа који учествују у тој кумулативној суми, а не учествују у парцијалној суми, односно кумулативној суми индекса *i* тј. *парцијална\_сума(l, d) = кумулативна\_сума(d) - кумулативна\_сума(l-1)*

1. Вредност елемента на датом индексу *i*: Тривијално се добија као парцијална сума од *i* до *i*. Последица ефикасности ове операције је да не морамо нужно у сваком тренутку да имамо физички сачуван низ већ конкретне вредности елемената можемо дохватати по потреби
2. Постављање вредности датог елемента низа на нову вредност *v2*. Прво израчунамо оригиналну вредност елемента, *v* коритећи четврту операцију и затим ажурирамо (операција 1) дати елемент на датом индексу прво за *v2– v*, а потом за.

Како последње три операције тривијално следе из прве две, у раду ће фокус бити само на првим двема.

**1. Технике решавања датог проблема**

**1.1 Наивне технике**

1. Класичан приступ:

Користимо обичан вектор имплементиран тако да има директан приступ сваком елементу (у даљем тексту ће се подразумевати таква имплементација вектора), који у потпуности одговара замишљеном низу из проблема.

* Ажурирање извршавамо једноставним сабирањем елемента са датом вредношћу.

*Временска сложеност просечног случаја: О(1)*

*Временска сложеност најнеповољнијег случаја: О(1)*

* За кумулативну суму пролазимо кроз све елементе од првог до датог индекса и сабирамо их.

*Временска сложеност просечног случаја: О(N/2)=O(N)*

*Временска сложеност најнеповољнијег случаја: О(N)*

1 4

1

1

1

1

1

2

3

4

5

+3

ажурирај (4, +3)

4

1

1

1

1

2

3

4

5

кумулативна сума (4)=1+1+1+4=7

*Слика 1.1: Класичан приступ*

Као што видимо овакав приступ јесте оптималан уколико знамо да нам је операција ажурирање много чешћа од потребе за кумулативним сумама, али је релативно спор уколико су и операције тражења кумулативних сума честе.

1. Оптимизација ка рачунању кумулативних сума:

Уместо да физички чувамо сам замишљени низ, чувамо вектор чији елементи представљају кумулативне суме на датим индексима.

* Ажурирање извршавамо тако што сваки елемент вектора од датог индекса па све до крајњег индекса (јер aжурирање елемента утиче на кумулативне суме тог елемента и свих после њега, а не утичe на оне пре) сабирамо са додатном вредношћу.

*Временска сложеност:*

*Средња сложеност: О(N/2)=O(N)*

*Сложеност најнеповољнијег случаја: O(N)*

* Кумулативну суму добијамо једноставним дохватањем елемента на одговарајућем индексу.

*Средња сложеност: О(1)*

*Сложеност најнеповољнијег случаја: O(1)*

1

1

1

1

1

1

2

3

4

5

замишљени низ:

ажурирај (4, +3)

4 7

5 8

3

2

1

1

2

3

4

5

+3

+3

7

8

3

2

1

1

2

3

4

5

кумулативна сума (4)= 7

*Слика 1.2: Оптимизација ка рачунању кумулативних сума*

Овај приступ је „инверзан“ у односу на први односно оптималан у специјалним случајевима када је друга операција много чешћа од прве, али такође спор уколико су обе операције честе.

Обе приступа имају меморијску сложеност *O(N)* пошто им је довољан један вектор исте величине каозамишљени низ.

Као што видимо ове „интуитивне“ технике имају своју примену у одређеним случајевима, али нису ефикасне у случајевима када су обе операције честе. Такође, како су тражене операције зависне једна од друге, немогуће је комбиновати ова два приступа над истим скупом операција.

**1.2 Фенвиково стабло**

Једна од структура података која ефикасно решава наш проблем, тј. обавља обе операције у логаритамској сложености јесте Фенвиково стабло, названо по Питеру Фенвику (*Peter Fenwick*) који је приказао како се дата структура података може искористити за аритметичко кодирање.

Прво, ради једноставности, сматраћемо да је замишљени низ индексиран тако да његов почетни елемент има индекс 1.

Фенвиково стабло је стабло у коме сваки елемент има свој индекс и вредност, а елементи са већим индексом су десно од елементат са мањим индексом. Број чворова једнак је броју елемената замишљеног низа уз један „вештачки елемент“ са индексом 0 и вредношћу 0 у корену стабла.

Синови корена стабла су елементи са индексима који су степени двојке (тј. имају само једну јединицу у бинарном запису). Свим осталим чворовима деца су они елементи чији индекси у свом бинарном запису имају још исти бинарни запис као тај чвор уз још тачно један бит са вредношћу 1 (у даљем тексу значајни бит) који се налази **након** најмањег значајног бита у том чвору (ако је индексу чвора нулти бит значајан или је једнак броју елемената низа, тај чвор је лист). На пример, у стаблу које одговара довољно великом низу, чвору са индексом *40 (1010002)* деца су чворови са индексима *41(1010012), 42 (1010102)* и *44 (1011002)*. Због овакве структуре стабла у литератури се ово стабло често и назива Бинарно индекстирано стабло (*Binary Indexed Tree*, такође често скраћено као *BIT* или *BITree*).

6

(0110)

4

(0100)

7

(0111)

5

(0101)

0

(0000)

2

(0010)

3

(0011)

1

(0001)

12

(1100)

9

(1001)

10

(1010)

11

(1011)

8

(1000)

*N* = 12

*Слика 1.3: Индекси у Фенвиковом стаблу*

Ради лакшег изражавања, уведимо релацију следбеник и њој инверзну релацију претходник. Следбеник датог чвора *u* је његов непосредни десни брат ако постоји или следбеник његовог оца ако такав следбеник постоји.

6

(0110)

4

(0100)

7

(0111)

5

(0101)

0

(0000)

2

(0010)

3

(0011)

1

(0001)

12

(1100)

9

(1001)

10

(1010)

11

(1011)

8

(1000)

*Слика 1.4: Следбеници чворова у стаблу*

Вредност чвора (осим нултог, који по дефиницији има вредност 0) у сваком тренутку једнака је суми вредности елемента у замишљеном низу који је на индексу који има и тај чвор и вредности његовог претходника, ако постоји. Уколико чвор нема претходника онда је његова вредност једнака вредности елемента замишљеног низа који му одговара.

замишљени низ:

3

2

1

1

2

1

2

3

4

5

3

6

7

8

6

5

4

7

8

9

10

11

9

12

6

(0110)

4

(0100)

7

(0111)

5

(0101)

0

(0000)

2

(0010)

3

(0011)

1

(0001)

12

(1100)

9

(1001)

10

(1010)

11

(1011)

8

(1000)

*Слика 1.5: Слика стабла које одговара датом замишљеном низу*

*Oперације:*

* Ажурирање се врши тако што се прво ажурира вредност чворa на датом индексу *i* за дату вредност *val*. Како је вредност тог чвора сабрана са *val*, како бисмо одржали својство стабла морамо сабрати и његовог следбеника са истом том вредности. Наравно, након тога морамо сабрати и следбениковог следбеника са *val* и тако итеративно понављати овај поступак докле год има следбеника у стаблу.

8

(1000)

12

(1100)

10

(1010)

9

(1001)

11

(1011)

6

(0110)

7

(0111)

4

(0100)

5

(0101)

3

(0011)

2

(0010)

0

(0000)

1

(0001)

+1

+1

+1

*Слика 1.6: Ажурирање чвора са индексом 3 за вредност +1.*

* Кумулативна сума индекса *i* је сума вредности свих чворова на путу од чвора са индексом *i* до корена стабла.

Како бисмо се уверили да да овакав алгоритам заиста даје кумулативну суму посматрајмо пут од датог индекса *i*. Прво битно опажање је да се сви индекси који учествују у његовој кумулативној суми налазе искључиво са леве стране у стаблу или на директном путу од њега до корена.

Такође, из поступка ажурирања закључујемо да ће у вредности тог чвора бити садржана вредност његовог парњака (овако ће на даље бити назван елемент у замишљеном низу чији индекс одговара индексу датог чвора) и вредности парњака све његове леве браће и **свих њихових потомака** (овај скуп чворова ће бити означен са *Pi*). Према поступку ажурирања тих елемената, након ажурирања чвора са индексом *i,* ажурира се његов следбеник, а он (ако уопште постоји) је сигурно десно од чвора са индексом *i* те сигурно неће бити његов предак. То значи да ће вредности свих парњака чворова из *Pi* бити тачно једном садржани у вредности чвора *i* и неће бити садржани ни у једном његовом претку. Слично, у вредности оца чвора *i* биће тачно по једном садржане вредности парњака његове леве браће и свих њихових потомака. Итеративним поступком увиђамо да ће на тај начин заиста вредност сваког елемента индекса мањег од *i* бити садржана тачно једном у вредности неког чвора на путу од чвора са индексом *i* до корена. Према томе, сабирањем вредности од чвора са индексом *i* до корена заиста добијамо кумулативну суму замишљеног низа за *i.*

6

(0110)

4

(0100)

7

(0111)

5

(0101)

0

(0000)

2

(0010)

3

(0011)

1

(0001)

12

(1100)

9

(1001)

10

(1010)

11

(1011)

8

(1000)

кумулативна сума (7)=4+5+8+0=17=2+1+2+3+2+3+4

*Слика 1.7: Кумулативна сума од индекса 7.*

Ради илустрације ових операција и како би читалац имао прилику још једном да прође кроз цео поступак, приказаћемо Фенвиково стабло са 5 елемената које одговара замишљеном низу {1,2,3,4,5} који је настао од почетног низа {0,0,0,0,0} редом ажурирајући елементе за вредности њихових индекса (тј. први за 1, други за 2 итд).

0

(000)

4

(100)

5

(101)

2

(010)

3

(011)

1

(001)

0)

0

(000)

4

(100)

5

(101)

2

(010)

3

(011)

1

(001)

1)

+1

+1

+1

+2

0

(000)

4

(100)

5

(101)

2

(010)

3

(011)

1

(001)

2)

+2

0

(000)

4

(100)

5

(101)

2

(010)

3

(011)

1

(001)

3)

+3

+3

0

(000)

4

(100)

5

(101)

2

(010)

3

(011)

1

(001)

4)

+4

0

(000)

4

(100)

5

(101)

2

(010)

3

(011)

1

(001)

5)

+5

0

(000)

4

(100)

5

(101)

2

(010)

3

(011)

1

(001)

6)

кумулативна сума (5)= 5+10+0=15=1+2+3+4+5

*Слика 1.8: Илустрација Фенвиковог стабла који одговара низу {1,2,3,4,5}*

Репрезентација стабла на рачунару:

Од репрезентације стабла очигледно зависи меморијска сложеност, али и временска (колико траје пут до оца или следбеника).

Непосредна идеја је да се користе показивачи/референце сваког чвора на синове и оца како би се директно симулирала структура стабла. Тада би се за налажење следбеника сваки пут морао обићи отац. То би се могло оптимизовати тако што за сваки чвор памтимо и његов показивач на његовог непосредног десног брата. Ипак, то би довело до још потрошње меморије и усложњавања кода те би имплементација била комплексна.

Погледајмо зато мало боље структуру стабла.

Синови датог чвора (осим чвора са индексом 0) су такви да њихови индекси имају исти бинарни запис као дати чвор осим што имају још тачно један значајни бит **након** свих значајних битова тог чвора. Према томе, може се рећи и обрнуто да је индекс неког чвора *ј* различит од индекса оца само у биту који се налази после очевих (тј. битова очевог индекса) односно у свом најмањем значајном биту. Према томе, за сваки чвор осим степена двојке индекс оца се добија простим уклањањем најмањег значајног бита његовог индекса. Чворовима индекса који су степени двојке (тј. имају само једну јединицу у бинарном запису) је отац чвор са индексом 0, што значи да се и за њих индекс оца може наћи на идентичан начин.

Уколико индекс неког чвора (поново ћемо занемарити степене двојке, па ћемо утврдити да ли је могуће употребити исту формулу) има значајни бит који није одмах „поред“ најмањег значајног бита оца, његовог следбеника тј. непосредног десног брата добијамо померањем његовог најмањег значајног бита за једно место у лево. Ова операција еквивалентна је сабирању индекса тог чвора са његовим најмањим значајним битом. Уколико чвор нема десног брата (тј. најмањи значајни бит му је одмах након најмањег значајног бита оца), а његов отац има десног брата, онда треба да обришемо (тј. поставимо на 0) најмањи значајни бит тог елемента, и померимо најмањи значајни бит оца за једно место. Сличан итеративни поступак имамо за сваки чвор, обришемо све узастопне значајне битове почев од најмањег значајног бита и први слободан бит (тј. први бит са вредношћу 0) поставимо на вредност 1. Увиђамо да је то увек еквивалентно сабирању вредности индекса са својим најмањим значајним битом. Како су чворовима чији су индекси степени двојке следбеници чворови чији су индекси следећи степени двојке (тј. бројеви са битом који су једно место лево), њихови следбеници се такође могу добити на описан начин. На овај начин рачунајући следбеника не морамо да губимо време приступајући оцу.

Према томе, како су за операције ажурирања и кумулативне суме једино неопходне релације оца и следбеника, ништа не губимо ако ово стабло представимо обичним вектором (таквим да имамо директан приступ произвољном индексу). Индекс чвора у стаблу ће одговарати индексу у вектору, а вредност чвора ће бити вредност елемента вектора. Према томе, као и код наивних приступа, просторна сложеност је *О(N+1)=O(N).*

*Временска сложеност операција:*

* Кумулативне суме:

Како у сваком кораку бришемо најмањи значајни бит све док је то могуће, време рачунања кумулативне суме датог индекса биће пропорционално броју значајних битова тог индекса.

*Сложеност најнеповољнијег случаја: О(logN)*

*Сложеност просечног случаја:* Нека је *m* најмањи број такав да се сви бројеви до *N* (рачунајући и *N*). Означимо са *d* просечан број посета чворовима тј. просечну дужину пута:

,

где је *pi* вероватноћа да неки број има тачно *i* значајних битова. Претпоставимо зарад једноставности да су сви бројеви од *m* битова (уз дозвољене водеће нуле) могући за индексе (а не само такви да су мањи од *N*). Оваквом претпоставком и даље остајемо на истој асимптотској сложености, а олакшавамо рачунање. Тада је *pi* једнак броју начина да се изабере *i* значајних битова од *m.* Тада сређивањем израза добијамо:



За сваки индекс  важи следеће:

Уколико је *m* паран, средњи члан израза је 

Према томе, упаривши одговарајуће сабирке у изразу добијамо:

Дакле, сложеност је *О(m/2)=O(logN/2)=O(logN),* при чему можда некад може бити од користи информација да је константан фактор приближан једној половини.

* Ажурирање:

Сличном анализом дошли бисмо и до сложености за ову операцију

*Сложеност најнеповољнијег случаја: О(logN)*

*Сложеност просечног случаја: О(logN)*

**2. Имплементација Фенвиковог стабла**

*Технички детаљи имплементације:*

1. Како се у каснијем делу рада упоређује Фенвиково стабло са наивним методама, ради лакшег упоређивања класе које представљају те три технике су направљене тако да имплементирају заједнички интерфејс (а то је могуће јер све користе један вектор исте величине).
2. Могуће је класе даље параметризовати тако да имплементирају било коју асоцијативну операцију за ажурирање и кумулативне суме и да вредности чворова буду било какав тип над којима је могуће извршити такве операције, али би таква параметризација (поготово операција), по мишљењу аутора, претерано одвукло пажњу од суштине рада, те су коришћене целобројне вредности и операција сабирања.
3. У окружењима које представљају означене целе бројеве као потпуне комплементе, супротан број датог целог броја је број који има исти најмањи значајни бит, а све остале битове супротне. Према томе, најмањи значајни бит се добија битовном конјукцијом броја и њему супротног броја. Таква имплементација је коришћена у раду.

У наставку је дата имплементација најважнијих операција (*v* је вектор који одговара стаблу, а *n* број елемената у структури) у језику *С++*. Конструктори и декларације метода се налази у коду који је приложен уз рад, али су изостављени из текста јер нису од посебног интереса за имплементацију саме структуре.

//pomocna metode

int FenvikovoStablo::najnizi\_znacajni\_bit(int broj)

{

return broj & (-broj);

}

//glavne metode

void FenvikovoStablo::azuriraj(int indeks,int dodatak)

{

while (indeks <=n)

{

v[indeks] += dodatak;

indeks += najnizi\_znacajni\_bit(indeks);

}

}

int FenvikovoStablo::kumulativna\_suma(int indeks)

{

int z = 0;

while (indeks>0)

{

z += v[indeks];

indeks -= najnizi\_znacajni\_bit(indeks);

}

return z;

}

**3. Упоређивање перформанси**

У овом делу рада биће илустрована брзина извршавања истог низа операција (над низом исте величине) коришћењем техника описаних у овом раду. Посматрани су низови величине 102, 104, 106 и 107 елемената. Операције (и вредности које су аргументи операција) биране су насумично, са параметризованим вероватноћама.

За врсту операције посматрани су случајеви кад су операције једнако вероватне тј. вероватноће су 50% – 50% за ажурирање и брисање, када је однос 70 – 30 или 30 – 70 (тј. једна операција је значајно заступљенија, али су обе честе) и када је однос 90 – 10 тј. 10 – 90 (једна операција далеко чешћа од друге). Операција је увек 107.

Величине аргумената операције су случајни бројеви до 100 са униформном расподелом.

Конкретна времена извршавања зависе од конфигурације на којима је покренут код. Ипак, ред величине времена ће остати сличан без обзира на конфигурацију (осим евентуално у случају супер-рачунара), као и односи времена извршавања различитих приступа који ће остати практично исти те се ове вредности могу сматрати ваљаном илустрацијом брзине алгоритама.

Уколико заинтересовани читалац покрене програм за тестирање из прилога, може му се учинити да времена нису одговарајућа (тј. да је разлика у извршавању програма и написаних времена извршавања операција већа него што би могло да се припише различитости хардверске конфигурације), али то је због тога што се у програму за тестирање користе скупе операције читања и уписивања у фајл, а нама су од интереса (и у овом раду су приложена) само времена извршавања операција над одговарајућом структуром података.

Резултати мерења:

Табеле 3.1: Вероватноћа ажурирања 10%

|  |  |  |  |
| --- | --- | --- | --- |
| Вероватноћа ажурирања: 10% | | | |
| Број елемената низа  (N) | Фенвик  (s) | Класичан приступ  (s) | Приступ оптимизован по кумулативним сумама  (s) |
| 10 | 0.119573 | 0.103866 | 0.103622 \* |
| 100 | 0.0949292 \* | 0.113767 | 0.0951767 |
| 1000 | 0.103212 \* | 0.166082 | 0.135863 |
| 10000 | 0.104227 \* | 0.759426 | 0.505727 |
| 100000 | 0.123703 \* | 6.79954 | 3.99421 |
| 1000000 | 0.212012 \* | 116.8 | 44.174 |
| 10000000 | 0.325937 \* | 2545.26 | 432.377 |

Легенда:

\* - најбоље време

време у интервалу [0, 1) s

време у интервалу [1, 100) s

време у интервалу [100, 1000) s

време у интервалу [1000, +∞) s

Табеле 3.2: Вероватноћа ажурирања 30%

|  |  |  |  |
| --- | --- | --- | --- |
| Вероватноћа ажурирања: 30% | | | |
| Број елемената низа  (N) | Фенвик  (s) | Класичан приступ  (s) | Приступ оптимизован по кумулативним сумама  (s) |
| 10 | 0.0959194 \* | 0.106055 | 0.098243 |
| 100 | 0.098075 \* | 0.106884 | 0.113107 |
| 1000 | 0.106645 \* | 0.185733 | 0.218937 |
| 10000 | 0.139868 \* | 0.60862 | 1.34428 |
| 100000 | 0.149107 \* | 5.1734 | 11.8731 |
| 1000000 | 0.209355 \* | 127.182 | 238.998 |
| 10000000 | 0.287475 \* | 1144.52 | 1150.49 |

Табеле 3.3: Вероватноћа ажурирања 50%

|  |  |  |  |
| --- | --- | --- | --- |
| Вероватноћа ажурирања: 50% | | | |
| Број елемената низа  (N) | Фенвик  (s) | Класичан приступ  (s) | Приступ оптимизован по кумулативним сумама  (s) |
| 10 | 0.102069 | 0.106363 | 0.0968525 \* |
| 100 | 0.100136 \* | 0.105295 | 0.116344 |
| 1000 | 0.10057 \* | 0.132974 | 0.295435 |
| 10000 | 0.117161 \* | 0.439051 | 1.99862 |
| 100000 | 0.13313 \* | 3.80119 | 19.7301 |
| 1000000 | 0.379962 \* | 108.675 | 247.804 |
| 10000000 | 0.392437 \* | 717.773 | 1840.46 |

Табеле 3.4: Вероватноћа ажурирања 70%

|  |  |  |  |
| --- | --- | --- | --- |
| Вероватноћа ажурирања: 70% | | | |
| Број елемената низа  (N) | Фенвик  (s) | Класичан приступ  (s) | Приступ оптимизован по кумулативним сумама  (s) |
| 10 | 0.0958366 | 0.093552 \* | 0.0941532 |
| 100 | 0.0975888 \* | 0.100418 | 0.123177 |
| 1000 | 0.115045 \* | 0.121436 | 0.421879 |
| 10000 | 0.111171 \* | 0.322113 | 2.90976 |
| 100000 | 0.1317 \* | 2.39044 | 27.9838 |
| 1000000 | 0.230146 \* | 35.8547 | 345.442 |
| 10000000 | 0.293738 \* | 457.648 | 2672.67 |

Табеле 3.5: Вероватноћа ажурирања 90%

|  |  |  |  |
| --- | --- | --- | --- |
| Вероватноћа ажурирања: 90% | | | |
| Број елемената низа  (N) | Фенвик  (s) | Класичан приступ  (s) | Приступ оптимизован по кумулативним сумама  (s) |
| 10 | 0.0991806 \* | 0.102061 | 0.104543 |
| 100 | 0.107671 | 0.106292 \* | 0.152684 |
| 1000 | 0.104752 \* | 0.110803 | 0.482047 |
| 10000 | 0.113972 \* | 0.171453 | 3.51003 |
| 100000 | 0.150972 \* | 0.88293 | 35.2736 |
| 1000000 | 0.278958 \* | 16.0647 | 561.354 |
| 10000000 | 0.275859 \* | 145.547 | 3318.73 |

Очекивано, времена извршавања операција на Фенвиковом стаблу не зависе значајно од врста операција пошто су операције исте асимптотске сложености (наравно времена не могу да буду потпуно идентична јер ипак зависе и од конкретних аргумената). Наивни приступи са друге стране драстично зависе од учесталости операција, што за исту (али велику) величину низа и број операција може направити разлику између пар минута и скоро сат времена. То значи да уколико погрешимо (или из неког разлога променимо) модел проблема те схватимо да је расподела проблема друкчија него што се очекивало (или друкчија него раније), морамо да у потпуности мењамо неки од наивних приступа другим приступом, док уколико смо почели да користимо Фенвиково стабло, можемо и да га задржимо чак и после промене.

Број операција мења времена свих приступа на исти начин (повећавање броја операција 10 пута би сва времена повећало отприлике 10 пута), што је и разлог зашто је у овом мерењу фиксиран. Са друге стране, величина низа би некоме ко не зна природу алгоритама већ само погледао податке деловала као да не утиче на време извршавања операција коришћењем Фенвиковог стабла (за разлику од наивних метода, на којима је приметна драстична разлика). Наравно, и величина низа утиче на време, али са логаритамским фактором, тј. тек би се увећавањем величине низа хиљаду пута време извршавања повећало десет пута. Дакле, како је време извршавања милион операција за величину низа 107 свега око 0.3s тек би се повећањем величине низа на огромних 1013 дошло до приметног успорења извршавања програма, а и тада би 30s било прихватљиво (мада би, нажалост, само читање толиког броја података са спољног извора трајало далеко дуже).

Примећујемо и да је, чак и у случају много чешће операције ажурирања него кумулативних сума, Фенвиково стабло значајно брже од класичног метода (осим за најмање низове). Вероватноћу операције ажурирања такву да је оправдано користити класичан метод уместо кумулативне суме за низ величине *N* налазимо на следећи начин:

Ta функција потребне вероватноће ажурирања изгледа на следећи начин на интервалу *(1,140]* (хоризонтална асимптота је *y=1*).



*Слика 3.1: функција потребне вероватноће ажурирања на интервалу (1,140]*

За низ величине 50 потребна је вероватноћа ажурирања око 90%, док је за још мање низове довољно чак и 70% (а са вероватноћом испод 67% никад не треба користити класичан приступ). Дакле, на малим низовима има смисла процењивати вероватноћу и размишљати о класичном приступу, мада ће тада тако малим низовима оба алгоритма бити изузетно брза.

Већ за *N=1000* потребна вероватноћа износи више од 99%. Тачна вероватноћа за сваку величину се лако може добити коришћењем бинарне претраге, али ће за „веће“ бројеве свакако бити изузетно блиска 1.

**4. Примене**

**4.1 Број инверзија низа**

Инверзија у неком низу *a* je пар индекса *i, ј* таких да је *ai > a*j и *i* < *j.* Како ефикасно пребројати број инверзија неког низа?

Како нам за број инверзија није потребна величина елемената у низу већ само њихов однос, можемо мапирати дати низ у низ чији су сви елементи бројеви између 1 и *N* тако да се најмањи елемент (и њему једнаки) мапирају у 1, следећи у 2 итд. Овакво операцију није тешко извести и уколико користимо неки ефикасан алгоритам сортирања, сложеност операције је *О(N)*.

Након тога можемо видети да је број инверзија у коме је дати елемент *ak* мањи од свог пара једнак броју већих елемената *ak* пре индекса *k* и тај број означити са *inv(k)*. Како је у свакој инверзији тачно један елемент мањи од другог, укупан број инверзија низа је .

Број бројева већих од *ak*је заправо збир броја појављивања броја *ak**+1* пре индекса *k*, броја појављивања *ak**+2* итд. све до броја појављивања вредности *N*. Број појављивања сваког броја може се ефикасно ажурирати у Фенвиковом стаблу, а збир појављивања бројева *[ak+1,N]* се ефикасно добија као *кумулативна\_сума(N)- кумулативна\_сума(ak)*.

Уколико је низ мапиран, временска сложеност једног рачунања *inv(k)* је *О(logN)*, а уколико није (што нећемо урадити ако су елементи већ мањи од *N*) *О(log(max(a))).*

Како је потребно извршити *N* онда је временска сложеност у случају мапирања *О(NlogN)* или *О(Nlog(max(a)))* без мапирања*.* Просторна сложеност је увек *О(N)* јер је свакако потребно чувати низ, али је већи константан фактор уколико су елементи „велики“ те је потребно извршити мапирање (јер је потребно чувати оригиналне позиције елемената пре сортирања).

*Псеудокод*

prebroj\_inverzije\_u\_nizu(niz)

{

N=velicina(niz)

fenvik.inicijalizuj(N)

brojac=0

for (indeks in (1..N))

{

fenvik.azuriraj(a[indeks],1)

brojac=brojac+fenvik.kumulativna\_suma(N)- fenvik.kumulativna\_suma(a[indeks])

}

return brojac;

}

Као што смо видели у овом примеру Фенвиково стабло се не користи искључиво када су нам операције ажурирања и кумулативне суме експлицитно задате већ се може користити и кад су нам потребне динамичке учесталости неких елемената.

**4.2 Рачунање интервала вероватноће догађаја**

Нека су дати догађаји *А1, А2… Аk* чије су вероватноће догађања *р1, р2… рk*.Уколико желимо да симулирамо извршавање ових догађаја помоћу случајне променљиве *u* униформне расподеле (0,1) то стандардно радимо на следећи начин:

* Уколико је , десио се догађај *А1*
* Уколико је , десио се догађај *А2*

*…*

* Уколико је , десио се догађај *Аk*

Ако немамо доступну расподелу ових вероватноћа већ само статистичку анализу редоследа извршавања догађаја и желимо да одредимо њихове вероватноће то можемо урадити релативно једноставно. Вероватноћа која одговара сваком од ових догађаја једнака је броју појављивања тог догађаја (коју ћемо означити са *ni*) подељена са укупним бројем догађаја (у ознаци *n*). На основу појединачних вероватноћа лако можемо одредити и интервале.

Међутим, шта ако нам је потребно да ове интервале одређујемо динамички, тј. кад год се деси догађај *Аi*да одредимо интервал променљиве *u* који одговара дотадашњој вероватноћи да се од тада деси догађај *Аi*.

Важно опажање је да се интервал који одговара догађају *Аi*може трансформисати на следећи начин



Овај облик нам омогућава да решимо проблем коришћењем Фенвиковог стабла. Приликом сваког догаћаја *Аi,* повећавамо (ажурирамо) променљиву *ni* за 1 и повећавамо укупан број догађаја (који чувамо у обичној бројачкој променљиви) за 1. За налажење интервала који одговарају догађају *Аi* потребне су нам кумулативна суме на индексима *i-1* и *i* подељене бројем *n.*

*Псеудокод*

odredi\_intervale\_za\_dogadjaje(maksimalni\_broj\_dogadjaja)

{

n=0

fenvik.inicijalizuj(maksimalni\_broj\_dogadjaja)

loop

{

dogadjaj=dohvatiDogadjaj()

n=n+1

fenvik.azuriraj(dogadjaj,1)

yield [fenvik.kumulativna\_suma(dogadjaj-1)/n, fenvik.kumulativna\_suma(dogadjaj) / n)

// da li je potreban cast u double zavisi od konkretnog jezika

}

}

Oвакав проблем се заправо среће у пракси код **аритметичког кодирања.**

Аритметичко кодирање је врста алогритма за компресију којим се садржај фајла мења децималним бројем (са нестандардно великим бројем децимала, али свакако значајно мањим од величине фајла) који одговара вероватноћама појављивања одређених симбола.

Уколико нпр. имамо знаке *А1, А2*, *А3 = ЕОF*којима одговарају редом статички интервали *[0,0.5), [0.5,0.7), [0.7,1)* и компресовани фајл са вредношћу 0.33 његов оригинални садржај се одређује на следећи начин:

Како је 0.33 на интервалу *[0,0.5)* први знак је *А1.*Затим како је 0.33 на интервалу *[0.25, 0.35)*, односно између 50% и 70% интервала *[0, 0.5)*, знамо да је следећи знак *А2.* На крају, пошто је 0.33 на интервалу *[0.25, 0.35)* односно на између 70% и 100% интервала *[0.25, 0.35)*, закључујемо да је следећи знак *А3*, који је уједно и последњи.

У варијантама овог алгоритма који користе динамичке вероватноће, за њихово брзо израчунавање користи се Фенвиково стабло. Заинтересовани читалац више о аритметичком кодирању може наћи у назив литературе.

**5. Вишедимензиона Фенвикова стабла**

**5.1 2D Фенвикова стабла**

Замислимо проблем сличан почетном. Имамо замишљену дводимензиону матрицу (са пољима индексираним од 1, ради једноставности) елемената и следеће 2 операције

1. Ажурирање елемента матрице за неки број

ажурирај (3, 3, +2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 8 |
| 3 | 9 | 10 | ~~11~~ 13 | 12 |
| 4 | 13 | 14 | 15 | 16 |

*Слика 5.1: Операција ажурурања у дводимензионом низу*

1. Кумулативна сума за поље *(i, j)*, тј. сума елемената подматрице чије је горње лево поље *(1, 1),* а доње десно *(i, j)*.

кумулативна\_сума (2, 2) = 1+2+5+6=14

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 | 4 |
| 2 | 5 | 6 | 7 | 8 |
| 3 | 9 | 10 | 13 | 12 |
| 4 | 13 | 14 | 15 | 16 |

*Слика 5.2: Кумулативна сума у дводимензионом низу*

Слично као у првом делу рада се из ових операција могу извести и преостале 3 операције.

1. Парцијална сума, тј. за дата поља *(i, j)* и *(k, l)* одредити збир елемената у подматрици у којој је поље *(i, j)* горње лево, а *(k, l)* доње десно.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | j |  | l |  |
|  |  |  |  |  |  |
| i |  |  |  |  |  |
| k |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

*Слика 5.3: 2D парцијалне суме*

Као што видимо са слике, наша парцијална сума је садржана у кумулативној суми за поље *(k, l)*. Међутим у тој суми су садржане и кумулативне суме за поља *(i-1, l)* и *(k, j-1).* Кад одузмемо те суме одузели смо кумулативну суму за поље *(i-1, j-1)* два пута, те га морамо додати једном. Према томе, формула за парцијалну суму је:

*парцијална\_сума(i, j) = кумулативна\_сума(k, l) - кумулативна\_сума(i-1, l) - кумулативна\_сума(k, j-1) + кумулативна\_сума(i-1, j-1)*

Како би формула давала одговарајући резултат и за граничне случајеве, када је нека од координата 1, додаћемо вештачки нулти ред и нулту колону у чијим су свим пољима вредности кумулативних сума 0.

Кумулативна сума за (*i, j*) се добија по формули:



Одмах примећујемо да се кумулативне суме (и ажурирање поља за њих) могу ефикасно пронаћи ако бисмо чували редове матрице у Фенвиковим стаблима.

Међутим и сума по редовима се такође увек тражи као сума узастопних елемената од почетног елемента до *i-*тог, односно тражи нам се сваки пут кумулативна сума кумулативних сума у редовима. Такве упите можемо ефикасно решити тако што и те редове организујемо у Фенвиково стабло, односно целу матрицу организујемо као Фенвиково стабло чији су чворови Фенвикова стабла.

Дакле, операције ћемо вршити на следећи начин:

* Ажурирање вршимо тако што ажурирамо *ј-*ти чвор у *i-*том Фенвиковом стаблу (на исти начин као у првом делу рада). Ажурирање тог чвора је утицало само на збирове кумулативних сума елемената **почев од елемента *ј* до краја,** те и код редова следбеника ажурирамо њихова Фенвикова стабла на елементу *ј***.**
* Кумулативне суме се рачунају прилично једноставно. Пролазимо кроз главно стабло по уобичајеном редоследу и за сваки чвор рачунамо његову кумулативну суму.

*Просторна сложеност:* Како се Фенвиково стабло може представити низом истог броја елемената као замишљени низ, тако се и Фенвиково стабло чији су чворови Фенвикова стабла може представљати низом *N* низова дужине *М* где су *N* и *М* број редова и колона матрице редом. Према томе, сложеност је *О(NМ)*

*Временска сложеност:*

Слично као што смо показали у једнодимензионом случају, сложености и просечног и најнеповољнијег случаја за обе операције је *О(logN ⋅ logM)*

*Имплементација:*

void Fenvik2D::azuriraj(int i, int j,int dodatak)

{

while (i <= N)

{

int pomj = j;

while (pomj <= M)

{

v[i][pomj] += dodatak;

pomj += najmanji\_znacajni\_bit(pomj);

}

i += najmanji\_znacajni\_bit(i);

}

}

int Fenvik2D::kumulativna\_suma(int i, int j)

{

int z = 0;

while (i > 0)

{

int pomj = j;

while (pomj > 0)

{

z+=v[i][pomj];

pomj -= najmanji\_znacajni\_bit(pomj);

}

i -= najmanji\_znacajni\_bit(i);

}

return z;

}

За ово стабло неће бити приказано поређење са наивним методама, али како је временска сложеност класичног приступа *О(NM),* разлика би била још драстичнија (а функција потребне вероватноће ажурирања за оправданост класичног приступа би се аналогно рачунала као  и била још ближа функцији *y=1*)

**5.2 D-димензионо (D>2) Фенвиково стабло**

Овај принцип се лако може уопштити. Уколико је потребно ажурирати елементе *D-*димензионог низа и рачунати кумулативне суме за поља *(i1, i2… iD)* то можемо учинити тако што одржавамо Фенвиково стабло (*D-1)-*димензионих Фенвикових стабала, на исти начин као што смо прешли са једнодимензионог на дводимензионе.

*Псеудокод:*

azuriraj(i1, i2...id,dodatak)

{

while (i1 <= N1)

{

pomi2 = i1;

while (pomi2 <= N2)

{

...

pomiD = iD;

while (pomiD <= ND)

{

v[i1][pomi2]...[pomiD]+=dodatak

iD+=najmanji\_znacajni\_bit(pomiD)

}

pomi(D-1)+= najmanji\_znacajni\_bit(pomi(D-1))

...

}

i1+= najmanji\_znacajni\_bit(pomi(i1 - 1))

}

}

kumulativna\_suma(i1, i2...id)

{

suma=0

while (i1 > 0)

{

pomi2 = i1;

while (pomi2 > 0)

{

...

pomiD = iD;

while (pomiD > 0)

{

suma += v[i1][pomi2]...[pomiD]

iD -= najmanji\_znacajni\_bit(pomiD)

}

pomi(D - 1) -= najmanji\_znacajni\_bit(pomi(D - 1))

...

}

i1 -= najmanji\_znacajni\_bit(pomi(i1 - 1))

}

return suma

}

*Сложеност:*

*Просторна:* *О(N1N2…ND)*

*Временска:* И просечна и најнеповољнија за обе операције *О(logN1 ⋅ logN2 ⋅ … ⋅ logND)*

*Вероватноћа операције ажурирања да би оправдала употребу класичног приступа:*



**6. Паралелизација операција над Фенвиковим стаблом**

Посматрајмо сада пример неколико узастопних операције ажурирања након које је позвана операција операција дохватања кумулативне сума

* ажурирај(индекс=1, додатак=2): повећавају се чворови на индексима 1, 2, 4, 8, 16, 32, … за 2
* ажурирај(индекс=5, додатак=3): повећавају се чворови на индексима 5, 6, 8, 16, 32, … за 3
* ажурирај(индекс=10, додатак=4): повећавају се чворови на индексима 10, 12, 16, 32, … за 6

0

(000000)

1

(000001)

2

(000010)

4

(000100)

8

(001000)

32

(100000)

16

(010000)

3

(000011)

5

(000101)

6

(000110)

7

(000111)

9

(001001)

10

(001010)

11

(001011)

12

(001100)

+1

+1

+1

+1

+1

+3

+3

+3

+3

+4

+4

+4

*Слика 6.1: Неколико узастопних ажурирања.*

Након ових операција елементи на индексима 1, 2, 4 су се повећали за 2, елементи на индексима 5 и 6 за 3, елементи на индексима 10 и 12 су се повећали за 6, елемент са индексом 8 се повећао за 2+3=5, а елементи са индексима који су степени двојке већи од 8 су се повећали за 2+3+6=11. Да се прво одиграла прво друга операција ажурирања, па прва па трећа, елементи на индексима 1, 2, 4 би се повећали за 2, елемент на индексима 5 и 6 за 3, елемент на индексу 8 за 3+2=5, а елементи на индексима већих степена двојке се повећавају за 3+2+6=11, дакле исти елементи се повећавају за исте вредности. Наравно, како је сабирање комутативна операција исти епилог би био и за трећу премутацију операција ажурирања. Такође, како није ни битно да ли је редослед операција ажурирања исти за различите елементе (нпр. елемент са индексом 8 може бити прво повећан за 2 (првом операцијом), а затим за 3, док елемент са индексом 16 може бити повећан за 3 па затим са 2 (и у било ком тренутку пре, после или између за 6), а потом елемент са индексом 32 може бити повећан за исте вредности у потпуно различитом редоследом од елемента са индексом 16). Јасно, ови закључци би важили за било који узастопни број операција ажурирања који долазе пре тражења кумулативне суме.

Према томе, све узастопне операције ажурирања које долазе пре операције тражења кумулативне суме се могу извршавати **паралелно** у различитим нитима уз једину синхронизацију да не могу различите операције истовремено ажурирати исти чвор како не би дошло до проблема са утркивањем.

Још је јасније да се блок узастопних операција кумулативних сума пре операције ажурирања може извршавати паралелно, с обзиром да оне уопште не мењају структуру стабла.

*Имплементација:*

Једна могућа имплеметација паралелног извршавања дата је у наставку (у прилогу уз рад је целокупан код).

void FenvikovoStabloParalelno::azuriraj\_paralelno(int indeks, int dodatak)

{

while (indeks <= n)

{

mutexi\_elemenata[indeks].lock();

v[indeks] += dodatak;

mutexi\_elemenata[indeks].unlock();

indeks += najnizi\_znacajni\_bit(indeks);

}

}

void FenvikovoStabloParalelno::kumulativna\_suma\_paralelno(int indeks,int &suma)

{

suma = 0;

while (indeks > 0)

{

suma += v[indeks];

indeks -= najnizi\_znacajni\_bit(indeks);

}

}

Један пример централне нити која синхронизује ове операције (врста операције и аргументи операције су енкапсулирани у структуру података *OperacijaSaArgumentima,* и оне су запамћене у одговарајућем вектору) тако да се извршавају истовремено нити са истом врстом операције, а затим кад заврше се извршава следећи блок са другом врстом операције итд.

double TestParalelno::vreme\_paralelno(std::vector<OperacijaSaArgumentima>& operacije, std::vector<int>& sume)

{

std::chrono::steady\_clock::time\_point pocetak = std::chrono::steady\_clock::now();

std::vector<std::thread> niti(broj\_operacija);

Operacija prethodna\_vrsta;

bool prvi = true;

int indeks\_sume = 0,pocetak\_bloka=0,kraj\_bloka,indeks\_operacije=0;

for (OperacijaSaArgumentima operacija : operacije)

{

if (!prvi && operacija.vrsta != prethodna\_vrsta)//izvrsi prethodne

{

for (int i=pocetak\_bloka;i<=kraj\_bloka;i++)

{

niti[i].join();

}

pocetak\_bloka = indeks\_operacije;

}

prethodna\_vrsta = operacija.vrsta;

kraj\_bloka = indeks\_operacije;

if (prvi)

prvi = false;

if (operacija.vrsta == AZURIRANJE)

{

niti[indeks\_operacije++]=move(std::thread(&FenvikovoStabloParalelno::azuriraj\_paralelno, &fenvik\_par, operacija.indeks, operacija.dodatak));

}

else

{

niti[indeks\_operacije++]=move(std::thread(&FenvikovoStabloParalelno::kumulativna\_suma\_paralelno, &fenvik\_par, operacija.indeks, std::ref(sume[indeks\_sume++])));

}

}

for (int i = pocetak\_bloka; i <= kraj\_bloka; i++)

{

niti[i].join();

}

std::chrono::steady\_clock::time\_point kraj = std::chrono::steady\_clock::now();

std::chrono::duration<double> trajanje = std::chrono::duration\_cast<std::chrono::duration<double>>(kraj - pocetak);

return trajanje.count();

}

Нажалост, макар на ауторовој конфигурацији, овакав начин извршавања је заправо спорији од секвенцијалног. Операције су саме по себи изузетно брзе и само прављење нити, промена контекста и синхронизација могу да успоре извршавање уколико није уложен велики труд у такву имплементацију. Питање је да ли је такав труд уопште оправдан у ситуацијама када је могуће радити секвенцијално с обзиром на брзину операција.

Ипак, добро је имати на уму да уколико је из неког другог разлога потребно имати више процеса или нити којима је само један од задатака извршавање неких операција над стаблом, ова структура података јесте погодна за паралелно извршавање операција над њом.

**КОРИШЋЕНИ АЛАТИ И ОКРУЖЕЊЕ**

За имплементацију, тестирање и мерење перформанси структурама података анализираних у овом раду коришћено је окружење *Microsoft Visual Studio 2019*, jeзик *С++ 11* на 64-битној верзији оперативног система *Windows 10*.

**ЗАКЉУЧАК**

Као што је показано, Фенвиково стабло је невероватно ефикасно за обављање операција ажурирања и кумулативних сума, а притом је и подједнако једноставно за имплементацију као и **наивни** методи.

Помоћу сегментних стабла и сличних структура података се ове операције могу извести са истом асимптотском временском сложеношћу, али Фенвиково стабло има мањи константни фактор у временској сложености, мању просторну сложеност (*О(N)* наспрам *O(NlogN)*) и једноставније је за писање кода.

У односу на наивне методе разлика у брзини је посебно приметна, док је једноставност кода на практично истом нивоу. Истина, уколико је једна операција много чешћа од друге, наивни методи могу бити бржи, али функција вероватноће једне операције такве да наивни метод оптимизован ка тој операцији показује да је већ за мало већу структуру података Фенвиково стабло бољи избор ако учесталост једне операције није изузетно близак 100%. Та разлика у перформансама је још уочљивија на вишедимензионим низовима.

Мора се признати да ове две операције (и оне које се тривијално могу извести из њих, као што је наведено на почетку рада) нису претерано честе у практичним применама. Ипак, с обзиром на изузетне перформансе и једноставност њене уопштивости и могућности паралелизације, корисно је бити свестан ове структуре података. Уколико се неки проблем може решити Фенвиковим стаблом (колико год то ретко било), Фенвиково стабло је вероватно најефикасније решење тог проблема.

**ЛИТЕРАТУРА**

1. Драган Стевановић, Марко Милошевић, Владимир Балтић, "Дискретна математика, основе комбинаторике и теорије графова, збирка решених задатака", ДМС, Београд, 2004
2. Војислав Петровић, "Проблем кенингсбершких мостова, почетак теорије графова", Тангента бр. 86/2, ДМС, Београд, 2016
3. Dmitri Fomin, Sergey Genkin, Ilia Itenberg, “Mathematical Circles (Russian Experience)”, AMS, 1996