УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Алекса Милојевић

СОРТИРАЊЕ ПОМОЋУ ТРАНСПОЗИЦИЈА

мастер рад

Београд, 2023.

Ментор:

др Миодраг Живковић, редован професор
Универзитет у Београду, Математички факултет

Чланови комисије:

проф др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

др Стефан Мишковић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: Септембар 2023.

Породици, девојци и пријатељима

Наслов мастер рада: Сортирање помоћу транспозиција

Резиме: Поређење генома у молекуларној биологији је битно за одређивање еволуције генома. Алгоритми са класичним поравнањем користе само локалне модификације, као што су *убацивање*, *брисање* и *замена* нуклеотида, занемаривајући глобалне модификације. Такви алгоритми су ограничено применљиви на геноме са великим бројем преуређења. Нека је $d(\pi)$ растојање задате пермутације π од идентичне пермутације I (минимални број транспозиција које π преводе у I). Пошто је одређивање транспозиционог растојања $d(\pi)$ јако тежак проблем, у раду су разматрани приближни алгоритми који одређују апроксимацију $\hat{d}(\pi)$ растојања $d(\pi)$. Такви приближни алгоритми могу се окарактерисати не само својом временском сложеношћу, него и односом апроксимације ρ, таквим бројем да за сваку пермутацију π важи неједнакост $\hat{d}(\pi)/d(\pi) \geq \rho$. У мастер раду разматрају се приближни алгоритми за одређивање транспозиционог растојања пермутације са односом апроксимације 1.75, односно 1.5. Ови алгоритми су програмски реализовани и приказан је резултат њихових израчунавања. На основу којих се може утврдити да алгоритам са односом апроксимације 1.5, сортира пермутацију са мањим бројем транспозиција, у односу на алгоритам са односом апроксимације 1.75.

Кључне речи: биоинформатика, молекуларна биологија, преуређење генома, транспозиције, симетрична група, приближни алгоритам

Садржај

Глава 1

Увод

Молекуларна биологија је научна дисциплина која се бави проучавањем биолошких процеса на молекуларном нивоу. Обухвата проучавање генетике, структуре и функције биомолекула, као и њихових интеракција у ћелијама и организмима. Један од циљева истраживања је да се боље објасни историја молекуларне еволуције врста. Интересантно је да иако се организација молекула различитих врста драстично разликује, сматра се да је садржај ДНК молекула често сличан између различитих врста. Преуређивање генома односи се на мутације које утичу на ту организацију. Свако проучавање преуређења генома укључује решавање проблема проналажења низа преуређења који трансформишу један геном у други.

Палмер и Хербон [2] су открили да се број операција потребних за трансформацију редоследа гена једног генома у други може користити као мера еволуционе удаљености између две врсте. Важно је напоменути да мера еволуционе удаљености две врсте не представља стварну еволуцију генома, већ даје доњу границу броја преуређења која су се десила и указује на сличност између две врсте. Међутим израчунавање мере еволуције није увек једноставно и не даје увек најбољу доњу границу.

На пример, геноми херпес вируса толико брзо еволуирају да је сличност између многих гена врло мала и не разликује се од позадинског шума.

Поређење секвенци на класичан начин није од користи за тако велике дивергентне геноме. Геноми еволуирају инверзијама и транспозицијама, као и уметањем, брисањем и трансформацијом фрагмената.

Проблеми који се у овом раду разматрају су поређење генома и анализа алгоритама за преуређење генома транспозицијама.

1.1 Пермутације

Пермутација секвенце од n бројева је преуређен редослед те секвенце. На пример, (2, 1, 3, 4, 5) је пермутација секвенце (1, 2, 3, 4, 5). Идентичка пермутација I реда n представља секвенцу бројева од 1 до n у растућем поретку, (1, 2 ... n). Битну улогу у поређењу генома играју геномски фрагменти или синтенски блокови. У преуређењу синтенског блока у геному, молекули у синтенском блоку не мењају поредак. Због тога се синтенски блок посматра као један елемент, а цео геном је пермутација синтенских блокова.

Проблем налажења секвенци операција преуређења за трансформацију једног генома у други може се посматрати као трансформација једне пермутације у другу. Проблем се може још поједноставити, као налажење секвенци операција преуређења једне пермутације у идентичку пермутацију. Поред тога, од интереса је налажење најкраће такве секвенце операција.

Проблеми који се односе на преуређење генома или сортирање пермутација су углавном доказано тешко решиви. Још увек су непознати, а мало је вероватно да ће икад бити пронађени, ефикасни полиномијални алгоритми. Зато су интересантни приближни алгоритми за решавање таквих проблема, који су релевантни само за проблеме који имају нумеричку циљну функцију (нпр. максимизација или минимизација проблема). У овом контексту проблем је апроксимација минимизације, која би омогућила да се види колико је алгоритам лошији од потенцијално савршеног алгоритма за најгори могући случај. Однос апроксимације је мера колико алгоритам може бити нетачан. Циљ је одредити алгоритам апроксимације са што бољом тачношћу, односно сложеношћу. Да би алгоритам апроксимације био користан, потребно је да укупно време извршавања алгоритма остане полиномијално.

1.2 Сортирање инверзијама

У истраживањима геномске структуре и еволуције често се користи сортирање инверзијама. Сортирање инверзијама представља промену редоследа сегмента у геному, окретањем и преслагањем сегмента у обрнутом редоследу у односу на њихово почетно стање, што омогућава да се испита како инверзије утичу на функцију гена, регулацију гена и развој организама. То омогућава и боље разумевање геномске организације и функције, као и механизама који

стоје иза еволуционих промена. У практичном смислу, сортирање генома инверзијама има различите примене. На пример, у генетичким истраживањима, сортирање инверзијама помаже у идентификацији и проучавању специфичних гена или региона генома. Такође, овај поступак је користан у развоју нових метода за детекцију генетских варијација или у проучавању генетских болести.

Геном може бити представљен пермутацијом $\pi = \pi_1\ \pi_2 \dots \pi_n$ са n различитих елемената, где је $1 \leq \pi_i \leq n$ за $1 \leq i \leq n$. Инверзија $\rho(i, j)$, за $1 \leq i < j \leq n$, делује на елементе $\pi_i \dots \pi_{j-1}$ пермутације π и трансформише је у пермутацију $\pi \times \rho = \pi_1 \dots \pi_{i-1}\ \pi_{j-1} \dots \pi_i\ \pi_j \dots \pi_n$.

На пример, ако је $\pi = 2\ 3\ 1\ 7\ 4\ 5\ 6$, онда је $\pi \times \rho(3, 6) = 2\ 3\ 4\ 7\ 1\ 5\ 6$.

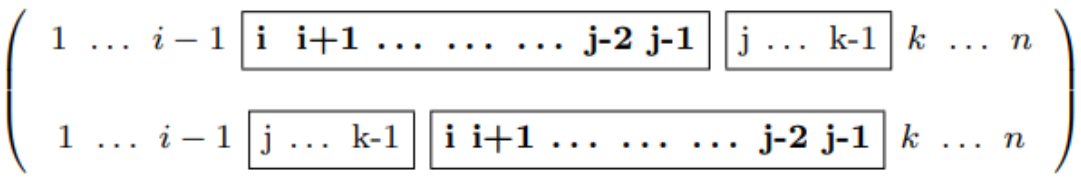
Растојање инверзије $d(\pi)$ између π и I представља најмањи број инверзија, таквих да је $\pi \times \rho_1 \times \dots \times \rho_{d(\pi)} = I$. Проблем сортирања инверзијама је проблем налажења најкраћег низа инверзија који трансформише пермутацију π у идентичку пермутацију I, односно одређивања растојања $d(\pi)$.

1.3 Сортирање транспозицијама

Иако је инверзија најчешћа операција преуређења, у овом раду се разматра транспозиција као операција преуређења. За транспозиције се верује да су најређе, али у случајевима са великим бројем преуређених генома (нпр. херпес вирус) транспозиција дугачких фрагмената уз инверзију делује као чести облик преуређења. Сортирање транспозицијама је тежи комбинаторни проблем од сортирања инверзијама.

Транспозиција $\rho = \rho(i, j, k)$, за неке индексе $1 \leq i < j \leq n+1$ и неко $1 \leq k \leq n + 1$ такво да $k \notin [i, j]$, пребацује елементе $\pi_i \dots \pi_{j-1}$ између π_{k-1} и π_k, чиме трансформише π у пермутацију $\pi \times \rho = \pi_0 \dots \pi_{i-1}\pi_j \dots \pi_{k-1}\pi_i \dots \pi_{j-1}\pi_k \dots \pi_{n+1}$. Другим речима, транспозиција $\rho = \rho(i, j, k)$ замењује места два узастопна блока $\pi_i \dots \pi_{j-1}$ и $\pi_j \dots \pi_{k-1}$, видети слику 1.1.

Транспозиционо растојање $d(\pi)$ између π и σ је најмањи број транспозиција $\rho_1, \rho_2, \dots$ таквих да је $\pi \times \rho_1 \times \dots \times \rho_{d(\pi)} = \sigma$. Проблем сортирања транспозицијама је налажење најкраће секвенце преуређења транспозицијом која трансформише пермутацију π у пермутацију σ. Транспозиционо растојање пермутација π и σ једнако је растојању транспозиције $\sigma^{-1}\pi$ и идентичке пермутације I. Сортирање пермутације π транспозицијама може се посматрати



Слика 1.1: Транспозиција

као проблем налажења транспозиционог растојања $d(\pi)$ између пермутација π и I. На пример, пермутација $\pi = 0\ 2\ 1\ 3\ 7\ 4\ 5\ 6\ 8$ се може сортирати применом две транспозиције, $\pi \times \rho(4,5,8) \times \rho(1,2,3) = I$, па је $d(\pi) = 2$:

$$0\ 2\ 1\ 3\ 7\ 4\ 5\ 6\ 8 \longrightarrow 0\ 2\ 1\ 3\ 4\ 5\ 6\ 7\ 8 \longrightarrow 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8.$$

Транспозиције генеришу симетричну групу S_n, те се свака пермутација може представити као композиција неких транспозиција. Проблем који се разматра је налажење најмањег броја t тако да је производ генератора $\rho_1 \times \rho_2 \times \cdots \times \rho_t$ једнак задатој пермутацији $\pi \in S_n$. Проблем налажења најкраћег производа генератора који је једнак π је јако тежак.

С обзиром да је проблем налажења транспозиционог растојања пермутације π јако тежак, потебно је разматрати приближне алгоритме за одређивање транспозиционог растојања пермутације. Приближни алгоритам треба да што ефикасније нађе резултат који приближно одговара решењу проблема. Однос апроксимације представља однос резултата добијеног приближним алгоритмом и решења проблема. Ако је $\hat{d}(\pi)$ резултат добијен приближним алгоритмом, а $d(\pi)$ решење проблема, тада је ρ однос апроксимације ако важи $\hat{d}(\pi)/d(\pi) \geq \rho$, за сваку пермутацију π.

У наставку рада разматрају се приближни алгоритми са односом апроксимације 1.75 и 1.5.

Глава 2

Алгоритми Бафна-Певзнер

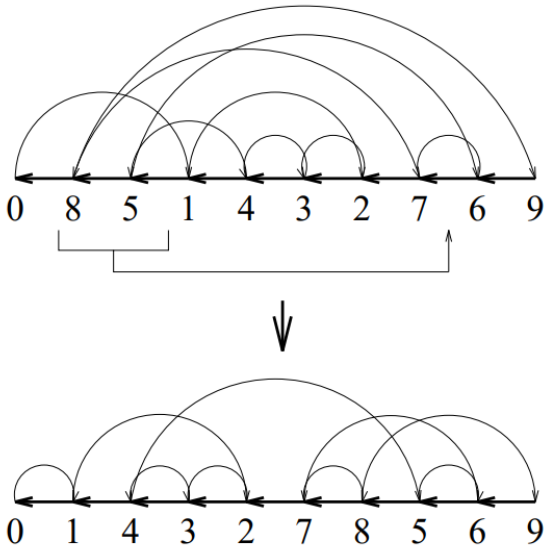
Ово поглавље приказује теоријске основе и карактеристике на којима се заснивају приближни алгоритми за сортирање транспозицијама. Алгоритми који ће бити приказани су алгоритми Бафна-Певзнер [1] са односом апроксимације 1.75 и 1.5.

2.1 Доња граница за сортирање транспозицијама

Пар (π_i, π_{i+1}) је *прекид*, у пермутацији π, ако важи да је $\pi_{i+1} \neq \pi_i + 1$, за $0 \leq i \leq n$. Може се приметити да је идентичка пермутација једина пермутација која нема прекиде, те се сортирање пермутације може посматрати као процес смањења броја прекида. Нека је укупан број прекида у пермутацији π означен са $\#breakpoints(\pi)$. Тада је $\#breakpoints(I) = 0$. Постоје специјални случајеви у којима пермутације са мањим бројем прекида имају веће растојање до идентичке пермутације, него пермутације са већим бројем прекида. Транспозиција може смањити број прекида у пермутацији за највише 3, чиме се добија доња граница $d(\pi) \geq \frac{\#breakpoints(\pi)}{3}$. Ипак, не дозвољавају све пермутације транспозицију која смањује број прекида за 3, тако да граница није довољно строга.

Да би се што тачније одредила доња граница, уводи се појам циклусног графа пермутације. Усмерени циклични граф пермутације π, означен са $G(\pi)$, је граф са скупом темена $\{0, 1, \ldots, n+1\}$ и скупом ивица дефинисан на следећи начин. За свако $1 \leq i \leq n+1$, сива (*директна*) ивица је од $i-1$ до i, црна

(*повратна*) ивица је од π_i до π_{i-1}. На слици 2.1 могу се видети примери повратних и директних ивица. Пример за повратну ивицу је ивица $(1, 5)$ која иде од 1 до 5, док је пример за директну ивицу ивица $(5, 6)$ која иде од 5 до 6. Чворови 1 и 6 могу се повезати ивицама $(1, 5)$ и $(5, 6)$, преко чвора 5.



Слика 2.1: Транспозиција $\rho(1, 3, 8)$ на π трансформише граф $G(\pi)$ у графу $G(\pi\rho)$

Алтернирајући циклус у графу $G(\pi)$ је циклус са наизменичним директним и повратним ивицама. Пошто се свака улазна ивица неког чвора, из графа $G(\pi)$, може упарити са излазном ивицом истог чвора различитог типа, постоји јединствена декомпозиција графа $G(\pi)$ у алтернирајуће циклусе. Алтернирајући циклус дужине *2k* означава се као *k-циклус*. Такође, *k-циклус* је *дуīачак* циклус ако је $k > 2$, у супротном је *краīак*.

На пример, на слици 2.1 у графу $G(\pi\rho)$ имамо:

- 1-циклусе: $(0, 1), (5, 6)$ и $(7, 8)$,
- 2-циклусе: $(1, 2, 3, 4)$ и $(3, 4, 1, 2)$,
- 4-циклусе: $(4, 5, 8, 9, 6, 7, 2, 3)$, $(2, 3, 4, 5, 8, 9, 6, 7)$, $(8, 9, 6, 7, 2, 34, 5)$ и $(6, 7, 2, 3, 4, 5, 8, 9)$.

У графу $G(\pi)$ има највише $n+1$ циклуса; идентичка пермутације је једина која има $n+1$ циклуса. За пермутацију π, број циклуса у графу $G(\pi)$ означава

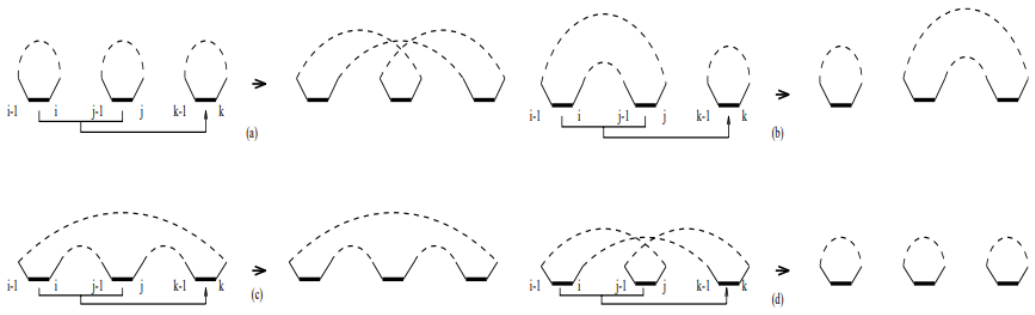
се са $c(\pi)$. Сортирање транспозицијама може се посматрати и као повећање броја циклуса пермутације π од $c(\pi)$ до $n+1$. Нека $\Delta c(\rho)$ означава промену броја циклуса у графу након транспозиције ρ, тада важи $\Delta c(\rho) = c(\pi\rho) - c(\pi)$.

Лема 1. $\Delta c(\rho) \in \{-2, 0, 2\}$.

Доказ. Како транспозиција $\rho(i, j, k)$ делује на шест чворова из графа $G(\pi)$, уклањајући три повратне ивице (π_i, π_{i-1}), (π_j, π_{j-1}) и (π_k, π_{k-1}) и додавајући три нове (π_j, π_{i-1}), (π_k, π_{j-1}) и (π_i, π_{k-1}). Уклоњене повратне ивице припадају једном, два или три различита циклуса. Тиме се добијају случајеви приказани на слици 2.2:

- Слика 2.2*a*: Уклоњене ивице припадале су различитим циклусима, а новододате истом циклусу, тада важи да је $c(\pi\rho) = c(\pi) - 3 + 1$.
- Слика 2.2*b*: Уклоњене ивице припадају у два циклуса, као и новододате, тада важи да је $c(\pi\rho) = c(\pi) - 2 + 2$.
- У случају када све уклоњене ивице припадају истом циклусу постоје два случаја:
  - Слика 2.2*c*: Ако новододате ивице припадају истом циклусу, тада важи да је $c(\pi\rho) = c(\pi) - 1 + 1$.
  - Слика 2.2*d*: Ако новододате ивице припадају различитим циклусима, тада важи да је $c(\pi\rho) = c(\pi) - 1 + 3$.

Добија се да је $\Delta c(\rho) \in \{-2, 0, 2\}$. □



Слика 2.2: Промена броја циклуса након примене транспозиције у цикличном графу

Тиме се добија нова доња граница за $d(\pi)$,

$$d(\pi) \geq \frac{n+1-c(\pi)}{2}.$$

Циклус у графу $G(\pi)$ је *непаран* ако има непаран број црних (*повратних*) ивица, у супротном је *паран*. Боља доња граница може се одредити ако се одвојено анализирају парни и непарни циклуси у графу. Како идентичка пермутација има само непарне циклусе, потребно је транспозицијама повећавати број непарних циклуса. Број непарних циклуса обележава се са $c_{odd}(\pi)$, а парних са $c_{even}(\pi)$. Са $\Delta c_{odd}(\rho) = c_{odd}(\pi\rho) - c_{odd}(\pi)$ се обележава промена броја непарних циклуса после примене транспозиције ρ.

Лема 2. $\Delta c_{odd}(\rho) \in \{-2, 0, 2\}$.

Доказ. Једнини случај када транспозиција креира више од два нова циклуса је случај описан на слици 2.2*d*. Ако су сва три новододата циклуса непарна, онда је и обрисан циклус био непаран, тада важи да је $c_{odd}(\pi\rho) = c_{odd}(\pi) - 1 + 3$, па је $\Delta c_{odd}(\rho) \leq 2$. На основу претходног тврђења и једнакости добија се да је $\Delta c_{odd}(\rho) \in \{-2, 0, 2\}$. □

Тиме се добија побољшана доња граница за $d(\pi)$,

$$d(\pi) \geq \frac{n+1-c_{odd}(\pi)}{2}.$$

Пример пермутације за коју се достиже побољшана доња граница, дат је у наставку. Нека је пермутација $\pi = 0, 2, 1, 3, 4, 5, 6$. Циклус $(0, 1, 2, 3, 1, 2)$ је 3-циклус, док су циклуси $(3, 4)$, $(4, 5)$ и $(5, 6)$ 1-циклуси, тада су сви циклуси у пермутацији π непарни. Укупан број непарних циклуса је четири, $c_{odd}(\pi) = 4$, тада је доња граница за израчунавање $d(\pi)$ једнака $\frac{n+1-c_{odd}(\pi)}{2} = \frac{5+1-4}{2} = 1$. Може се приметити да транспозиција $\rho(1, 2, 3)$ сортира пермутацију π, тиме се добија да је $1 = d(\pi) \geq \frac{n+1-c_{odd}(\pi)}{2} = 1$.

2.2 Горња граница за сортирање транспозицијама

Транспозиција ρ је k-потез ако важи да је $\Delta c(\rho) = k$, за $k \in \{-2, 0, 2\}$. Да би сортирање било брже, потребно је да се користи што је више могуће

2-потеза. У овом раду детаљније се разматрају циклуси који омогућавају 2-потезе.

Ако i означава ивицу од π_i до π_{i-1}, онда транспозиција $\rho(i, j, k)$ делује на ивице i, j и k, као и на циклус C ако све три ивице, i, j и k, припадају циклусу C.

На основу леме 1 постоје два запажања за транспозицију ρ.

- Ако делује на један циклус и креира више од једног новог циклуса у графу $G(\pi\rho)$, онда је ρ 2-потез (слика 2.2*d*).
- Ако делује на ивице које припадају у тачно два различита циклуса, онда је ρ 0-потез (слика 2.2*b*).

Посматра се k-циклус C који редом садржи повратне ивице $i_1, \ldots, i_k$, такав циклус се може записати на k различитих начина у зависности од избора првог елемента. Нека је први елемент најдешњи елемент у циклусу $i_1 = \max_{1 \leq t \leq k} i_t$. За свако $k > 1$ циклус C је *неоријентисан* ако је $i_1 \ldots i_k$ опадајући низ, у супротном C је *оријентисан*. За неоријентисане циклусе дефинише се и смер ивица. Директна ивица која спаја $\pi_t = i - 1$ са $\pi_s = i$ је *лево* оријентисана ако је $t > s$, у супротном је *десно* оријентисана. Циклус C је неоријентисан ако и само ако је $k > 1$ и C има тачно једну десну ивицу и то између i_k и i_1.

Лема 3. *Неоријентисани циклуси не дозвољавају 2-потез, док оријентисани циклуси дозвољавају 2-потез.*

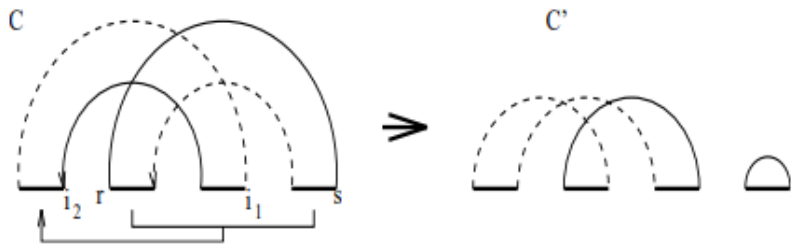
Доказ. Нека је циклус $C = (i_1, \ldots, i_k)$ оријентисан циклус и нека је $3 \leq t \leq k$ индекс такав да је $i_t > i_{t-1}$. Посматра се транспозиција $\rho(i_{t-1}, i_t, i_1)$ која делује на циклус C, транспозиција креира 1-циклус $(\pi_{i_{t-1}-1}, \pi_{i_t})$ и неке друге циклусе, следи да је ρ 2-потез. □

На основу претходне леме, може се увести следећа теорема.

Теорема 1. *За сваку пермутацију различиту од идентичке постоји или 2-потез или 0-потез праћен 2-потезом.*

Доказ. Ако граф $G(\pi)$ садржи оријентисан циклус, тада је могућ 2-потез. У супротном, се посматра неоријентисан циклус $C = (i_1, \ldots, i_k)$ у графу $G(\pi)$ и нека је r позиција највећег елемента у интервалу $[i_2, i_1 - 1]$ пермутације π, а s позиција од $\pi_r + 1$ у π. Може се уочити да је $\pi_s > \pi_r$, па важи $s \notin$

$[i_2, i_1]$. Претпоставља се да је $s > i_1$, посматра се транспозиција $\rho(r+1, s, i_2)$ (слика 2.3). Пошто транспозиција делује на два различита циклуса, према претходном тврђењу, транспозиција ρ је 0-потез и мења правац леве ивице (π_{i_1-1}, π_{i_2}), али не мења правац десне ивице (π_{i_k-1}, π_{i_1}). У графу $G(\pi\rho)$ циклус C' садржи бар две десне ивице, (π_{i_1-1}, π_{i_2}) и (π_{i_k-1}, π_{i_1}), следи да је циклус ограничен и да постоји 2-потез. □



Слика 2.3: Креирање оријентисаног циклуса 0-потезом

Са два узаступна потеза, укупан број циклуса ($c(\pi)$) може се повећати за најмање 2 ($\Delta c(\rho_1\rho_2) \geq 2$). Из ове чињенице следи горња граница за сортирање транспозицијама: свака пермутација π се може сортирати применом $n + 1 - c(\pi)$ транспозиција. Тиме је изведен приближни алгоритам за сортирање транспозицијама са гарантованим односом 2.

Добијена горња граница се не ослања на -2-потезе, већ само на 0-потезе и 2-потезе. Наравно, могуће је да оптималан алгоритам користи и -2-потезе.

2.3 Укрштајући циклуси

Показано је да је број 2-потеза већи или једнак од броја 0-потеза. Да би се даље повећале перформансе за сортирање транспозицијама потребно је додатно повећати број 2-потеза, за то је неопходно увести додатне дефиниције.

Посматра се тројка повратних ивица (x, y, z) које припадају истом циклусу C. С обзиром да постоје три различита записа за ову тројку, потребно је изабрати онај који почиње са најдешњом повратном ивицом, $\max\{x, y, z\}$, за репрезентацију тројке (x, y, z). Тројка је *неоријентисана* ако важи $x > y > z$, у супротном је *оријентисана*. Све тројке неоријентисаног циклуса су

неоријентисане, док сваки оријентисан циклус има бар једну оријентисану тројку.

Уређене секвенце $\{v_1 < \cdots < v_k\}$ и $\{w_1 < \cdots < w_k\}$ су укрштене ако важи $v_1 < w_1 < v_2 < w_2 < \cdots < v_k < w_k$ или $w_1 < v_1 < w_2 < v_2 < \cdots < w_k < v_k$. Скупови V и W су укрштени, ако су и сортирани редоследи скупова V и W укрштени. Нека је (x, y, z) неоријентисана тројка, нпр. $x > y > z$. Транспозиција $\rho(i, j, k)$ је транспозиција *преуређења* у односу на тројку (x, y, z) ако се тројке (i, j, k) и (x, y, z) укрштају.

Нека је (x, y, z) тројка из C и нека су i, j и k повратне ивице које не припадају C. Тада транспозиција $\rho(i, j, k)$ мења оријентацију тројке (x, y, z) ако и само ако је транспозиција ρ транспозиција преуређења за (x, y, z).

За сваку тројку $(x, y, z) \in C$, где је циклус C неоријентисан, транспозиција $\rho(z, y, x) = \rho(y, x, z)$ трансформише циклус C у оријентисан циклус.

Транспозиција $\rho(y, z, x) = \rho(z, x, y)$ је 2-потез, ако је тројка (x, y, z) оријентисана.

Циклуси C и C' се укрштају, ако постоји оријентисана тројка у C и неоријентисана тројка у C' које се укрштају. Циклуси C и C' се не укрштају ако постоје оријентисане тројке и у C и у C' које се не укрштају.

У пермутацији π постоје два узастопна 2-потеза, ако пермутација π има укрштене или неукрштене циклусе.

Транспозиција делује на циклусе C и C' ако делује на повратне ивице оба циклуса.

Нека је C циклус који садржи повратне ивице x и y и нека је D циклус који садржи повратне ивице x' и y'.

Нека је ρ транспозиција која делује на три од четири повратне ивице x, y, x' и y'.

- Ако се x, y не укрштају са x', y', онда ρ креира циклус са неоријентисаном тројком.
- Ако се x, y и x', y' укрштају, онда ρ креира циклус са оријентисаном тројком.

За циклус $C = (i_1, \ldots, i_k)$ се каже да обухвата циклус $D = (j_1, \ldots, j_l)$, ако важи $i_k < j_l < j_1 < i_1$.

За сваки неоријентисани циклус $C = (\dots, a, \dots, b, \dots)$ са произвољним ивицама a, b, постоји циклус $D = (\dots, c, \dots, d, \dots)$ такав да се (a, b) и (c, d) укрштају.

Теорема 2. *Ако постоји дуг циклус у графу $G(\pi)$, онда су у пермутацији π могући или 2-потез или 0-потез праћен са два узастопна 2-потеза.*

Доказ. Ако $G(\pi)$ има оријентисан циклус, онда је 2-потез могућ. Ако постоје неоријентисани дуги циклуси C и D са укрштеним тројкама $(r, s, t) \in C$ и $(x, y, z) \in D$, онда је 0-потез ρ који делује на ивице z, y, x транспозиција преуређења за C. Транспозиција ρ трансформише C у оријентисани циклус C', а D у D'. Примећује се да се C' и D' укрштају, те постоје два узастопна 2-потеза у графу $G(\pi\rho)$.

Претпоставља се да нема два циклуса чије се тројке укрштају. Бира се неоријентисан дуги циклус $C = (i_1, \dots, i_k)$ и налази се циклус са повратним ивицама c и d тако да се парови (c, d) и (i_1, i_k) укрштају. Нека је то циклус $D = (x, \dots, c, \dots, d, \dots, y)$. Ако је $y < i_k$, онда је $x < i_1$, у супротном би D обухватало C. Ако је $y > i_k$, онда је $x > i_1$, у супротном се (c, d) и (i_1, i_k) не укрштају. Тада важи $y < i_k < x < i_1$ или $i_k < y < i_1 < x$, претпоставља се да важи други случај.

Нека је s најдешња ивица у C, лево од y, $\max_{i \in C, i<y} i$, тада постоје два случаја.

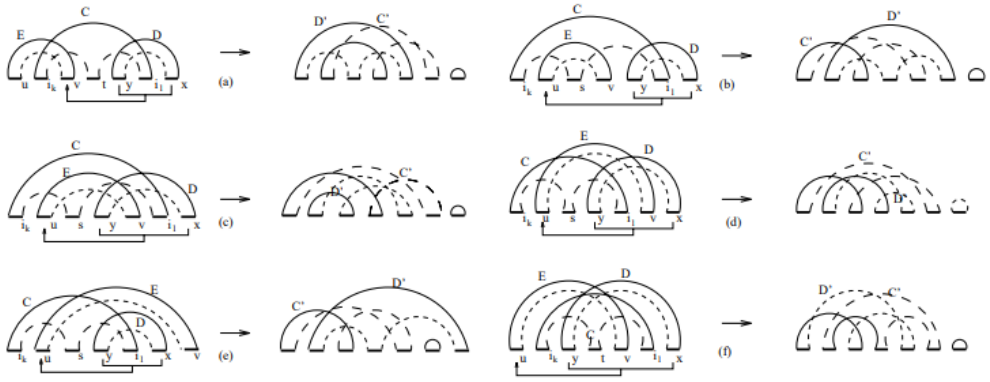
- Први случај: $s > i_k$.

  Тражи се циклус $E = (v, \dots, c, \dots, d, \dots, u)$ такав да се (c, d) и (i_k, s) укрштају. Ако је $u < i_k$, онда је $v < s$, у супротном E обухвата C (ако $v > i_1$) или има тројку која се укршта са $(i_k, s, i_1) \in C (s < v < i_1)$. Ако је $u > i_k$, тада постоје четири случаја у зависности ком сегменту припада v ($[s, y]$, $[y, i_1]$, $[i_1, x]$ или $[x, n+1]$). Транспозиције $\rho(x, y, v)$ (слика 2.4*a*) и $\rho(x, y, u)$ (слике 2.4*b* − *e*) су транспозиције преуређења у односу на тројку (i_1, s, i_k) из C, чиме трансформише C у оријентисан циклус C'. Такође трансформише D и E у D' и 1-циклус у графу $G(\pi\rho)$. На слици 2.4*a* D' је оријентисан, док је у осталим случајевима оријентисан када је $v \in [y, x-1]$. На слици 2.4*b* и слици 2.4*e* C' и D' се укрштају, на слици 2.4*c* и слици 2.4*d* се не укрштају. У оба случаја два узастопна 2-потеза су могућа у графу $G(\pi\rho)$.

- Други случај: $s = i_k$.

  Нека је t најлевља повратна ивица из C десно од y, $t = \min_{i \in C, i > y} i$. Како је C дугачак циклус, важи $t < i_1$. Тражи се циклус $E = (v, \ldots, c, \ldots, d, \ldots, u)$ тако да се парови (c, d) и (t, i_1) укрштају. Циклуси E и D се разликују, јер би у супротном E и C имали тројке које се укрштају. Ако је $v > i_1$, онда $u > t$, у супротном би циклус E обухватао C (за $u < i_1$) или имао тројку које се укршта са $(i_1, t, i_k) \in C$ (ако $i_1 < u < t$)). Ако је $v < i_1$, онда постоје три случаја у зависности ком сегменту припада u ($[0, i_k]$, $[i_k, y]$ или $[y, t]$). Први случај је приказан на слици 2.4f, док су друга два представљени слици 2.4c и слици 2.4e. На слици 2.4f, транспозиција $\rho(x, y, u)$ трансформише C у неоријентисан циклус C', а D и E у оријентисан циклус D' и 1-циклус у графу $G(\pi\rho)$. Добија се да се C' и D' укрштају, те су могућа два узастопна 2-потеза у графу $G(\pi\rho)$.

□



Слика 2.4: 0-потез праћен 2-потезом

Овим се долази до алгоритма за сортирање транспозицијама са односом апроксимације 1.75.

2.4 Алгоритам за сортирање транспозицијама са односом апроксимације 1.75

Када граф $G(\pi)$ има дуге циклусе, на основу теореме 2 може се применити 2-потез или 0-потез праћен са два узастопна 2-потеза. Алгоритам нам гарантује креирање најмање четири циклуса у три потеза, када граф има дуг циклус, чиме се у просеку добија $\Delta c(\rho) = \frac{4}{3}$, што је боље него $\Delta c(\rho) = 1$. У случају када $G(\pi)$ има само кратке циклусе, на основу теореме 1 може се применити 0-потез праћен 2-потезом чиме се креирају четири 1-циклуса од два 2-циклуса у два потеза. Тада 0-потез праћен 2-потезом пружа у просеку $\Delta c(\rho) = \frac{4-2}{2} = 1$. Али, за ова два потеза се достиже максимална оцена за креирање непарних циклуса $\Delta c_{odd}(\rho) = \frac{4-0}{2} = 2$. Први део алгоритма нам и даље не гарантује да је $\Delta c_{odd}(\rho) = 2$ за сваки 2-потез. Без обзира да ли се користи укупан број циклуса или укупан број непарних циклуса, не може се гарантовати оцена боља од 2. Побољшана гаранција перформанси се добија комбинованом циљном функцијом која у зависности од парних и непарних циклуса даје побољшан однос апроксимација.

Алгоритам *Tsort(π)*:

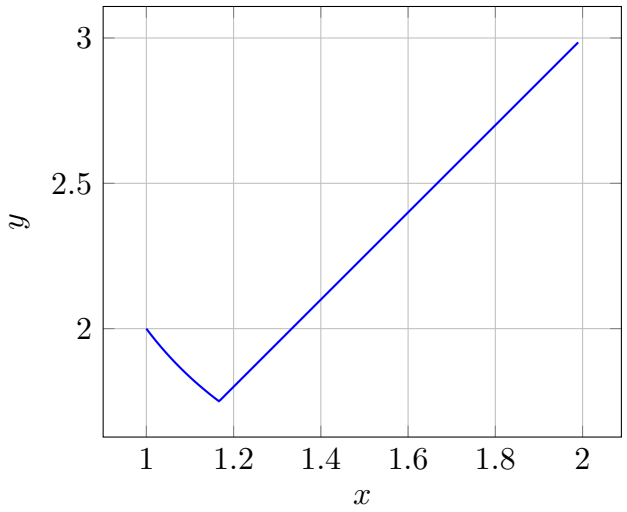
- Докле год $G(\pi)$ има дуге циклусе, примени 2-потез или 0-потез праћен са два узастопна 2-потеза (теорема 2).
- Ако $G(\pi)$ има само кратке циклусе, примени 0-потез праћен 2-потезом (теорема 1).

Tsort остварује однос апроксимације од 1.75 за сортирање транспозицијама.

Доказ. Нека је, за $x \geq 1$, дефинисана функција $f(\pi) = xc_{odd}(\pi) + c_{even}(\pi)$. За тако задато x, $f(\pi)$ је јединствено максимизирана идентичком пермутацијом, а сортирање транспозицијама одговара максимизирању функције f. Максимум која нека транспозиција ρ може да оствари је $\Delta f(\pi) = f(\pi\rho) - f(\pi) = 2x$. На основу претходних теорема представља се максимум за Δf.

Ако граф садржи само кратке циклусе, онда се након два потеза креирају четири 1-циклуса од два 2-циклуса, чиме се добија $4x - 2$ за два потеза или у просеку $2x - 1$ за једну транспозицију. Сваким 2-потезом, два нова циклуса се

креирају и у најгорем случају (ако су оба парна) може се добити $\Delta f = 2$. 0-потез или креира 1-циклус или не мења број повратних веза у циклусу. Тиме се добија да је $\Delta f \geq 0$ за сваки 0-потез. Сваки 0-потез праћен са два узастопна 2-потеза у просеку остварује $\Delta f = \frac{4}{3}$. Спајањем два случаја, када у графу има дуги циклус и када нема, у просеку се добија $\Delta f \geq \min\{\frac{4}{3}, 2x-1\}$. У поређењу са најбољим могућим случајем добијеним *Tsort* алгоритмом, добија се однос апроксимације $\frac{2x}{\min\{\frac{4}{3},2x-1\}}$. Најбољи однос апроксимације се достиже за $x = \frac{7}{6}$, приказано на графу 2.5, тиме се добија 1.75 за однос апроксимације. □



Слика 2.5: Графички приказ односа апроксимације за функцију Δf

Имплементација овог алгоритма дата је у одељку Прилог.

2.5 Алгоритам за сортирање транспозицијама са односом апроксимације 1.5

Потребно је побољшати претходни алгоритам, да би однос апроксимације био још бољи. За то је потребно повећати број непарних циклуса, јер идентичка пермутација има $n + 1$ циклус дужине један. Избором одговарајућег 2-потеза повећава се број непарних циклуса за два сваким 2-потезом.

Транспозиција ρ је *валидна* ако важи $\Delta c(\rho) = \Delta c_{odd}(\rho)$. За циклус C који садржи ивице i и j, нека је $d(i, j)$ број повратних веза између π_i и π_j. На пример, ако важи да је $d(i, j) = 1$ онда су i и j узаступне повратне ивице.

Ако у графу $G(\pi)$ постоји оријентисан циклус онда или постоји валидан 2-потез или валидан 0-потез праћен са два узастопна 2-потеза.

Циклус који не дозвољава валидан 2-потез је оријентисан циклус са комплексном самоукрштајућом структуром. Потребно је избегавати креирање таквих циклуса, за то је потребно дефинисати *строго оријентисан* циклус, који има једноставну самоукрштајућу структуру међу свим оријентисаним циклусима.

Нека је $C = (i_1, \ldots, i_k)$ циклус и нека је $C^* = (i_1 = j_1 > \cdots > j_k)$ секвенца повратних веза C у опадајућем низу. Секвенце C и C^* се поклапају за неоријентисани циклус, у супротном су различити. Дефинише се *строго оријентисан* циклус као оријентисан циклус ако се C^* може трансформисати у C једном транспозицијом. На пример, нека је $C = C_1C_2C_3C_4$ и $C^* = C_1C_2C_3C_4$ (C_4 може бити празно); запажа се да се C^* може трансформисати у C једном транспозицијом.

Сваки строго оријентисан циклус има тачно две десно оријентисане ивице, али нису сви оријентисани циклуси са две десно оријентисане ивице строго оријентисани.

Строго оријентисани циклус дозвољава валидан 2-потез.

Ако је ρ транспозиција преуређења на неоријентисан циклус C, онда ρ трансформише C у строго оријентисан циклус.

Нека су $D(x, \ldots, y)$ и $E = (x', \ldots, y')$ два неоријентисана циклуса без укрштајућих тројки и нека је ρ транспозиција која делује на три од четири повратне ивице x, y, x' и y'. Тада ρ креира строго оријентисан циклус ако и само ако D и E имају укрштајуће парове ивица.

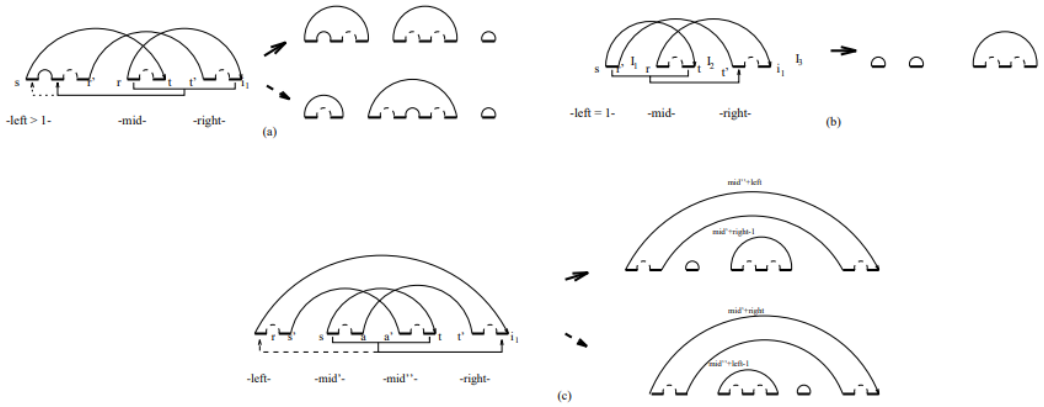
Сваки строго оријентисан циклус има тачно две повратне ивице, нека је прва (r, i_1), а друга (s, t). На слици 2.6*a*, за строго оријентисан циклус прве врсте важи

$$r' = \max_{i \in left} i \text{ и } t' = \min_{i \in right} i,$$

и могу се посматрати интервали $I_1(C) = [r', r]$, $I_2(C) = [t, t']$ $I_3(C) = [0, s] \cup [i_1, n+1]$. За строго оријентисане циклусе друге врсте, слика 2.6*c*, важи

$$s' = \max_{i \in left} i,\ t' = \min_{i \in right} i,\ a = \max_{i \in mid'} i \text{ и } a' = \min_{i \in mid''} i$$

и могу се посматрати интервали $I_1(C) = [s', s]$, $I_2(C) = [t, t']$ $I_3(C) = [a, a']$.



Слика 2.6: Строго оријентисани циклуси: $(a), (b)$ прве врсте. (c) друге врсте

Строго оријентисан циклус C и неоријентисан циклус $C' = (i_k, \ldots, i_k)$ се *строго укрштају* ако постоји повратна ивица x у C' тако да сваки сет $I_1(C)$, $I_2(C)$ и $I_3(C)$ садржи тачно један од елемената тројке (i_1, x, i_k).

Ако граф $G(\pi)$ има строго укрштајући циклус, онда постоје два узастопна валидна 2-потеза у графу $G(\pi)$.

Транспозиција ρ је *сигурна* у односу на строго оријентисан циклус $C \in G(\pi)$, ако трансформише C у строго оријентисан циклус у графу $G(\pi\rho)$.

Нека је C строго оријентисан циклус и нека је $(x, y, z) \notin C$ тројка таква да се ни једна ивица из C не налази између x и y. Тада је транспозиција која делује на (x, y, z) сигурна у односу на C.

Нека су циклуси C и C' строго оријентисани. Циклус C је строго неукрштајући циклус у односу на C' ако има десно оријентисану ивицу (a, b) такву да се ни једна повратна веза из C' не налази у интервалу $[a, b]$.

Ако $G(\pi)$ има строго неукрштене циклусе, онда постоје два узастопна валидна 2-потеза у графу $G(\pi)$.

Теорема 3. *Ако постоји дупачак циклус у графу $G(\pi)$, тада постоји или валидан 2-потез или валидан 0-потез праћен са два узастопна валидна 2-потеза у π.*

Доказ. Опонаша се доказ теореме 1 уз додатан услов да су сви потези валидни.

Ако $G(\pi)$ има оријентисан циклус, онда је могуће применити валидан 2-потез или валидан 0-потез праћен са два узастопна валидна 2-потеза. У случају када $G(\pi)$ има неоријентисане циклусе C и D са тројкама које се укрштају $(r, s, t) \in C$ и $(x, y, z) \in D$. Тада $\rho(x, y, z)$ трансформишу C у строго оријентисан циклус C' у графу $G(\pi\rho)$ и трансформише D у неоријентисан циклус D' у графу $G(\pi\rho)$. Посматрају се интервали $I_1(C')$, $I_2(C')$ и $I_3(C')$ такве да сваки садржи тачно један елемент неоријентисане тројке у D'. Због тога се C' и D' строго укрштају и два узастопна валидна 2-потеза су могућа у графу $G(\pi\rho)$.

Разматра се случај када граф $G(\pi)$ нема оријентисане циклусе или циклусе са тројкама које се укрштају, тада постоји шест могућих случаја, видети слику 2.4:

- Слика 2.4*a*:

  Посматра се валидан 0-потез $\rho(x, y, v)$ трансформише циклусе D и E у неоријентисан циклус D' и 1-циклус, а циклус C трансформише у строго оријентисан циклус C' у графу $G(\pi\rho)$. Може се приметити да π_x, π_v и π_y припадају D' и $\pi_y \in I_1(C')$, $\pi_v \in I_2(C')$ и $\pi_x \in I_3(C')$, што значи да се C' и D' строго укрштају. Следи да су могућа два узастопна валидна 2-потеза у графу $G(\pi\rho)$.

- Слика 2.4*b*:

  Посматра се валидан 0-потез $\rho(x, y, u)$ трансформише циклусе D и E у неоријентисан циклус D' и 1-циклус, а циклус C трансформише у строго оријентисан циклус C' у графу $G(\pi\rho)$. Може се приметити да $\pi_y \in I_1(C')$ и $\pi_u \in I_2(C')$. Затим се бира v као најдешња ивица лево од y, тако да ни једна ивица из C није између v и y, и важи $\pi_v \in I_3(C')$. Пошто се у D' налазе π_x, π_v и π_y, циклуси C' и D' се строго укрштају. Следи да су могућа два узастопна валидна 2-потеза у графу $G(\pi\rho)$.

- Слика 2.4*c*:

  Посматра се валидан 0-потез $\rho(u, v, x)$ који трансформише циклус C у строго оријентисан циклус C' и трансформише циклусе D и E у строго оријентисан циклус D' и 1-циклус. Затим се дефинише a као најдешњу ивицу у D лево од i_1 ($a = \max_{i \in D, i < i_1} i$) и b као најлевљу ивицу у C десно од y ($b = \min_{i \in C, i > y} i$). Може се приметити како мора да важи $a < b$, јер

у супротном би се тројке $(i_k, b, i_1) \in C$ и $(y, a, x) \in D$ укрштају. Ако је $b > v$, онда не постоји ивица из C у интервалу $[v, i_1]$, а самим тим ни у C' не постоји повратна ивица која се налази у сегменту који покривају повратне ивице $(\pi_{y-1}, \pi_x) \in D'$. Према томе D' је строго неоријентисан у односу на C'. Ако важи да је $a < b < v$, онда не постоји повратна ивица из D у интервалу $[v, i_1]$, ни повратне ивице из D' која се налази у сегменту који покривају повратне ивице $(\pi_{i_k-1}, \pi_{i_1}) \in C'$. Према томе C' је строго неоријентисан у односу на D'. У оба случаја могућа су два узастопна валидна 2-потеза.

- Слика 2.4d:

  Посматра се валидан 0-потез $\rho(x, y, u)$ трансформише циклусе D и E у строго оријентисан циклус D' и 1-циклус, а циклус C трансформише у строго оријентисан циклус C' у графу $G(\pi\rho)$. Нека је a најдешња ивица у E лево од s и нека је b најлевља ивица у E десно од i_1. C нема ивицу између ивица $b \in E$ и $x \in D$ у графу $G(\pi)$ пошто важи $i_1 < b < x$. Такође C нема ни ивицу e у интервалу $[u, a]$, јер у супротном би се $(u, a, v) \in E$ укрштала са $(e, s, i_1) \in C$. у графу $G(\pi\rho)$, C' нема ни једну повратну ивицу у секвенци коју обухватају повратне ивице $(\pi_{b-1}, \pi_a) \in D'$, следи да је D' строго неоријентисан у односу на C'. Према томе могућа су два узастопна валидна 2-потеза.

- Слика 2.4f:

  Посматра се валидан 0-потез $\rho(x, y, u)$ трансформише циклус C у строго оријентисан циклус C', а циклусе D и E у циклус D' и 1-циклус у графу $G(\pi\rho)$. Циклус D' је строго оријентисан ако D и E имају парове који се укрштају, у супротном је неоријентисан. Ако је D' оријентисан циклус, онда је исти случај као у прошлом случају. Ако је D' неоријентисан, онда се посматра π_y, π_u, π_v из D' и нека су $\pi_y \in I_1(C')$, $\pi_u \in I_2(C')$ и $\pi_v \in I_3(C')$, следи да се C' и D' строго укрштају. Према томе постоје два узастопна валидна 2-потеза у графу $G(\pi\rho)$.

- Слика 2.4f:

  Посматра се валидан 0-потез $\rho(x, y, u)$ трансформише циклусе D и E у строго оријентисан циклус D' и 1-циклус, а циклус C трансформише у неоријентисан циклус C' у графу $G(\pi\rho)$. У наставку се посматра

$\pi_{i_1}, \pi_t, \pi_{i_k} \in C'$ које се налазе редом у $I_2(D')$, $I_1(D')$ и $I_3(D')$. Следи да се C' и D' строго укрштају. Према томе постоје два узастопна валидна 2-потеза у графу $G(\pi\rho)$.

□

Претходна тврђења важе када граф $G(\pi)$ има дуге циклусе. За кратке циклусе потребан је другачији приступ.

За 0-потез се каже да је *добар*, ако повећава број непарних циклуса за два.

Теорема 4. *Ако Граф $G(\pi)$ има само кратке циклусе, могуће је применити добар 0-потез праћен валидним 2-потезом.*

Доказ. Опонаша се доказ теореме 1. 0-потезом на два 2-циклуса креира се оријентисан 3-циклус и 1-циклус, чиме се повећава број непарних циклуса за два. Стога, 0-потез је добар. Након чега се може применити валидан 2-потез на оријентисан 3-циклус. □

На основу претходних тврђења, могуће је дефинисати алгоритам *TransSort* у ком ће сви потези бити валидни или добри.

Алгоритам *TransSort(π)*:

- Докле год $G(\pi)$ има дуге циклусе, примени валидан 2-потез или валидан 0-потез праћен са два узастопна валидна 2-потез (теорема 3).
- Ако $G(\pi)$ има само кратке циклусе, примени добар 0-потез праћен валидним 2-потезом (теорема 4).

Сложеност алгоритма *TransSort* за сортирање транспозицијама је $O(n^2)$. Алгоритам *TransSort* сортира пермутацију π у највише $\frac{3}{4} \cdot (n + 1 - c_{odd}(\pi))$ транспозиција, тиме се добија однос апроксимације 1.5.

Имплементација овог алгоритма дата је у одељку Прилог.

Глава 3

Програмска реализација и резултати израчунавања

У овом поглављу приказују се програмска реализација и резултати извршавања алгоритама описаних у одељцима 2.4 и 2.5.

3.1 Програмска реализација

Генерише се насумична пермутација дужине n, која се сортира помоћу аглоритама *Tsort* и *TransSort*.

Програм примењује алгоритме *Tsort* и *TransSort* за сортирање пермутације и приказује број транспозиција потребних да би се пермутација сортирала.

Програм је писан у програмском језику $C\#$ окружење $.NET6$.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace SortingByTranspositions
{
    internal static class SortingByTranspositions
    {
        static void Main(string[] args)
        {
            SortedDictionary<int, int> permutationTsort;
            SortedDictionary<int, int> permutationTransSort;
```

```
Dictionary<int, List<List<int>>>
    dictionaryPermutation = new Dictionary<int, List<
    List<int>>>();

for (int j = 6; j < 36; j++)
{
     int n;
     if (j < 20)
         n = j;
     else
         n = (int)Math.Pow(1.1635, j);

     int seed = 3;
     List<List<int>> randomPermutations = new List<
        List<int>>();
     for (int i = 0; i < 10; i++)
     {
         List<int> randomPermutation;
         do
         {
             randomPermutation =
                GenerateRandomPermutation(n, seed);
             seed++;
         }
         while (randomPermutations.Contains(
            randomPermutation));

         randomPermutations.Add(randomPermutation);
     }

     dictionaryPermutation.Add(n, randomPermutations)
        ;
}

foreach (var listPermutations in
    dictionaryPermutation)
{
```

```
Console.WriteLine(listPermutations.Key);

foreach (var randomPermutation in
   listPermutations.Value)
{
    permutationTsort =
       ConvertPermutationFromList(
       randomPermutation);
    permutationTransSort =
       ConvertPermutationFromList(
       randomPermutation);
    int numberOfOddCycles =
       TransSortImplementation.NumberOfOddCycles
       (permutationTransSort);
    double minDistance = (permutationTransSort.
       Count - numberOfOddCycles - 1) / 2;

    Console.WriteLine("Donja granica za
       sortiranje transpozicijama je: " +
       minDistance);

    permutationTsort.Tsort(out int distanceTsort
       );

    if (permutationTsort.IsSorted())
    {
        Console.WriteLine("Permutacija je
           sortirana algoritmom 'Tsort'.");
        Console.WriteLine("Broj potrebnih
           transpozicija za sortiranje je: " +
           distanceTsort);
    }

    permutationTransSort.TransSort(out int
       distanceTransSort);

    if (permutationTransSort.IsSorted())
```

```
                {
                    Console.WriteLine("Permutacija je
                        sortirana algoritmom 'TransSort'.");
                    Console.WriteLine("Broj potrebnih
                        transpozicija za sortiranje je: " +
                        distanceTransSort);
                }
            }
        }
    }

    private static bool IsSorted(this SortedDictionary<int,
        int> permutation)
    {
        foreach (KeyValuePair<int, int> kvp in permutation)
            if (kvp.Key != kvp.Value)
                return false;

        return true;
    }

    public static List<int> GenerateRandomPermutation(int n,
        int seed)
    {
        List<int> numbers = Enumerable.Range(1, n).ToList();
        Shuffle(numbers, seed);

        List<int> result = new List<int> { 0 };
        result = result.Concat(numbers).ToList();
        result.Add(n + 1);
        return result;
    }

    public static void Shuffle(List<int> list, int seed)
    {
        Random rng = new Random(seed);
        int n = list.Count;
```

```
            while (n > 1)
            {
                n--;
                int k = rng.Next(n + 1);
                int value = list[k];
                list[k] = list[n];
                list[n] = value;
            }
        }


        // Od permutacije pravimo strukturu uredjenih parova,
           kako bi dobili listu parova (i, pi)
        public static SortedDictionary<int, int>
           ConvertPermutationFromList(List<int> permutationList)
        {
            SortedDictionary<int, int> result = new
               SortedDictionary<int, int>();

            for (int i = 0; i < permutationList.Count; i++)
                result.Add(i, permutationList[i]);

            return result;
        }
    }
}
```

3.2 Резултати истраживања

У истраживању је коришћен узорак на 10 насумично одрабраних пермутација за сваку дужину. Резултати истраживања приказани су у табели 3.1, где су у колонама приказане вредности:

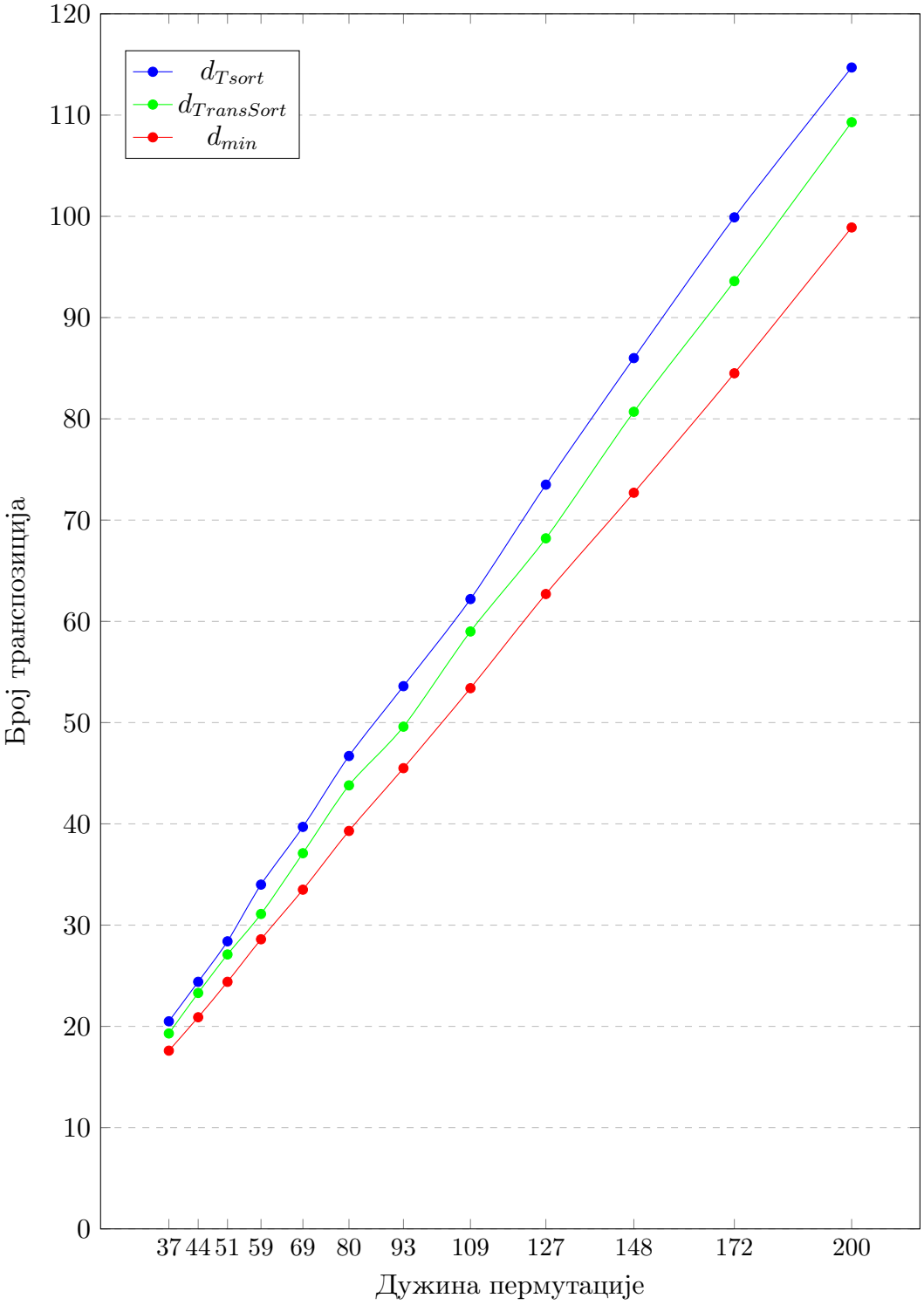
- дужина пермутација,
- просечна доња граница за сортирање пермутација,
- просечан број транспозиција потребних за сортирање пермутација алгоритмом $Tsort$,
- просечно време извршавања алгоритма $Tsort$,
- просечан број транспозиција потребних за сортирање пермутација алгоритмом $TransSort$,
- просечно време извршавања алгоритма $TransSort$.

На дијаграму на слици 3.2 приказане су просечне вредности за доњу границу сортирања пермутација транспозицијама и просечан број транспозиција потребних за сортирање пермутација алгоритмима $Tsort$ и $TransSort$. На дијаграму на слици 3.3 приказане су вредности просечног одступања алгоритама $Tsort$ и $TransSort$ од доње границе сортирања. Алгоритам $Tsort$ има лошији однос апроксимације у односу на алгоритам $TransSort$, стога користи већи број транспозиција за сортирање пермутација.

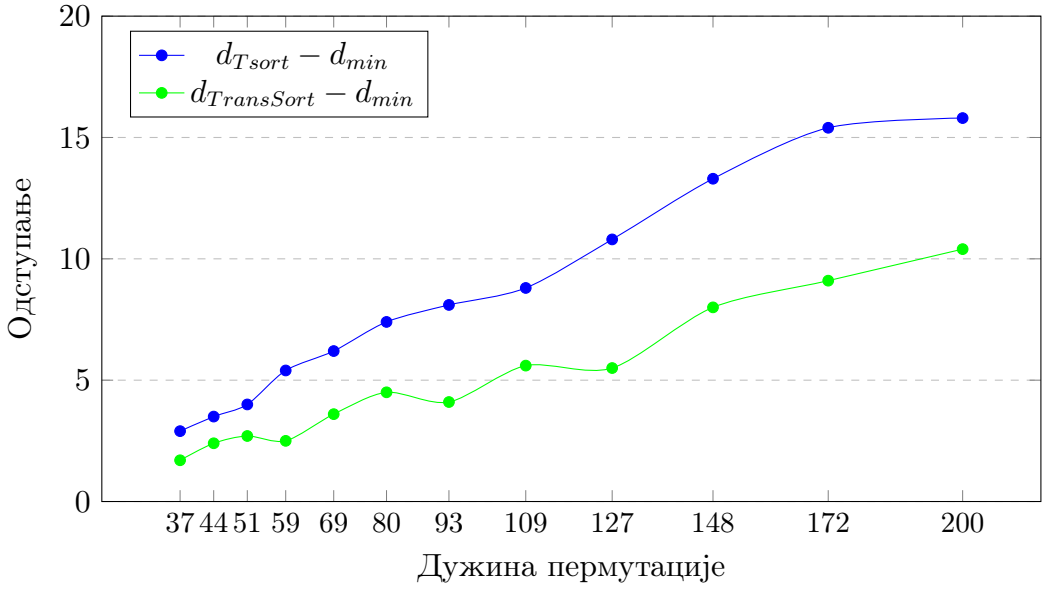
На дијаграму на слици 3.4 приказан је логаритам просечног времена извршавања алгоритама $Tsort$ и $TransSort$. У складу са очекивањем, алгоритам који пермутације сортира применом мањег броја транспозиција се у просеку дуже извршава.

n	$d_{min}(\pi)$	$d_{Tsort}(\pi)$	$t_{Tsort}(\pi)$	$d_{TransSort}(\pi)$	$t_{TransSort}(\pi)$
6	2.2	2.5	0.0029s	2.3	0.0041s
7	2.7	3.0	0.0002s	2.8	0.0027s
8	3.3	3.9	0.0014s	3.5	0.0021s
9	3.4	4.5	0.0002s	4.0	0.0007s
10	4.3	4.7	0.0004s	4.5	0.0007s
11	4.8	5.1	0.0012s	4.9	0.0024s
12	5.2	5.6	0.0009s	5.4	0.0024s
13	5.8	6.4	0.0013s	6.1	0.0024s
14	6.3	8.3	0.0009s	6.6	0.0021s
15	7.0	9.2	0.0020s	7.3	0.0044s
16	7.3	8.7	0.0018s	8.0	0.0056s
17	7.8	9.4	0.0028s	8.5	0.0061s
18	8.5	9.7	0.0028s	9.0	0.0105s
19	8.5	9.8	0.0031s	8.8	0.0114s
20	9.2	10.9	0.0031s	9.9	0.0078s
24	11.7	13.6	0.0045s	12.8	0.0127s
27	13.0	15.0	0.0053s	14.0	0.0174s
32	15.1	17.9	0.0063s	16.6	0.0166s
37	17.6	20.5	0.0110s	19.3	0.0365s
44	20.9	24.4	0.0172s	23.3	0.0633s
51	24.4	28.4	0.0259s	27.1	0.1126s
59	28.6	34.0	0.0372s	31.1	0.1561s
69	33.5	39.7	0.0647s	37.1	0.4991s
80	39.3	46.7	0.1254s	43.8	1.5061s
93	45.5	53.6	0.1870s	49.6	2.5479s
109	53.4	62.2	0.2888s	59.0	3.2586s
127	62.7	73.5	0.4917s	68.2	4.8273s
148	72.7	86.0	0.8793s	80.7	19.2121s
172	84.5	99.9	1.5573s	93.6	34.9778s
200	98.9	114.7	2.9305s	109.3	95.6799s

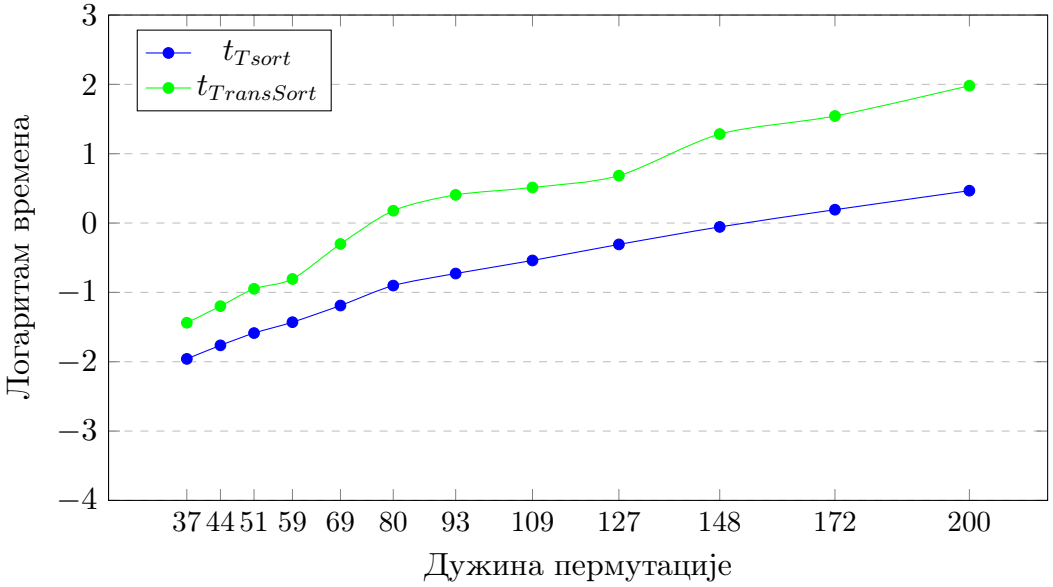
Слика 3.1: Нумерички резултати истраживања



Слика 3.2: Просечан број транспозиција потребних за сортирање пермутације



Слика 3.3: Просечно одступање алгоритама $Tsort$ и $TransSort$ од доње границе сортирања



Слика 3.4: Логаритам просечног времена извршавања алгоритама $Tsort$ и $TransSort$

Глава 4

Закључак

Преуређивање генома може се извршити убацивањем, брисањем, заменом, фузијом, фисијом, транслокацијом, инверзијом или транспозицијом. У овом раду разматрају се само транспозиције, без обзира што се сматра да су инверзије чешће мутације од транспозиција.

Сортирање пермутације транспозицијама омогућује да се закључи колика је разлика између два генома настала временом. У раду су приказани приближни алгоритми за одређивање апроксимације $\hat{d}(\pi)$ транспозиционог растојања $d(\pi)$ задате пермутације π од идентичне пермутације I (минимални број транспозиција које π преводе у I). Пошто је одређивање транспозиционог растојања $d(\pi)$ јако тежак проблем, у раду су приказани приближни алгоритми за одређивање транспозиционог растојања са границом апроксимације 1.75, односно 1.5. Ови алгоритми су програмски реализовани. Интересантан могући правац даљег рада је реализација приближног алгоритма из рада [3], који има границу апроксимације 1.375 и много мању временску сложеност $O(n \log n)$ у поређењу са реализованим алгоритмима.

Библиографија

[1] *Sorting by Transpositions*, V.Bafna and P.A.Pevzner. In: Siam J. Discrete Math, Vol. 11, No. 2, pp. 224–240, May 1998.

[2] *Advances in Genome Rearrangement Algorithms*, M.Hasan and M.S.Rahman. In: Algorithms in Computational Molecular Biology: Techniques, Approaches and Application, M.Elloumi and A.Y.Zomaya, pp. 749-761, 2011.

[3] *The 1.375 Approximation Algorithm for Sorting by Transposition Can Run in $O(n \log n)$ Time*, J.S.Firoz, M. Hasan, A.Z. Khan, and Md.S.Rahman. In: WALCOM: Algorithms and Computation, Md.S.Rahman and S.Fujita, pp 161-166, February 2010

Прилог

У овом одељку приказане су имплементације алгоритама за сортирање транспозицијама описаних у одељку 2.

Алгоритам *Tsort*

Имплементација алгоритма *Tsort* описаног у одељку 2.4. Пример покретања програма налази се у одељку 3.1, а резултати израчунавања су приказани у табели 3.1 у одељку 3.2.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace SortingByTranspositions
{
    internal static class TsortImplementation
    {
        // Sortiranje permutacije Bafna-Pevzner algoritmom
           koriscenjem 0-poteza i 2-poteza
        // Ulaz: Pocetna permutacija
        // Out argument: Rastojanje pocetne permutacije do
           identicke
        // Izlaz: Sortirana permutacija
        public static void Tsort(this SortedDictionary<int, int>
            permutation, out int count)
        {
            count = 0;
            while (!permutation.IsSorted())
            {
```

```
            // Na orijentisane duge cikluse primenjujemo 2-
               potez
            if (permutation.GetLongOrientedCycle(out
               Dictionary<int, int> longOrientedCycle))
                permutation.MakeTwoMove(longOrientedCycle,
                   ref count);

            // Na neorijentisane duge cikluse primenjujemo
               0,2,2-potez
            else if (permutation.HasLongCycle())
                permutation.MakeZeroTwoTwoMove(ref count);

            // Na kratke cikluse primenjujemo 0,2-potez
            else if (permutation.GetTwoCycle(out Dictionary<
               int, int> twoCycle))
            {
                if (!twoCycle.IsCycleNonoriented())
                    permutation.MakeTwoMove(twoCycle, ref
                       count);
                else
                    permutation.MakeZeroTwoMove(twoCycle,
                       ref count);
            }
        }
    }

    // Primenjujemo 0,2-potez na 2-ciklus u permutaciji
    private static void MakeZeroTwoMove(this
       SortedDictionary<int, int> permutation, Dictionary<
       int, int> twoCycle, ref int count)
    {
        int a = twoCycle.ElementAt(1).Key;
        int b = twoCycle.ElementAt(0).Key;
        int max = 0;
        int r = 0;

        for (int i = a; i < b; i++)
```

```
        {
            KeyValuePair<int, int> element = permutation.
                ElementAt(i);
            if (element.Value > max)
            {
                max = element.Value;
                r = element.Key;
            }
        }

        permutation.TryGetValue(r, out int pi_r);
        int s = permutation.First(el => el.Value == pi_r +
            1).Key;

        if (s >= a && s <= b)
            return;

        if (s > b)
            permutation.Transposition(r + 1, s, a, ref count
                );
        else
            permutation.Transposition(a, b, s, ref count);

        List<Dictionary<int, int>> cycles = permutation.
            DecomposePermutation();
        Dictionary<int, int> cycleForTwoMove = cycles.First(
            cycle => cycle.ContainsKey(a));

        permutation.MakeTwoMove(cycleForTwoMove, ref count);
    }

    // Primenjujemo 2-potez na ciklus u permutaciji
    // Ulaz: permutacija, ciklus
    private static void MakeTwoMove(this SortedDictionary<
        int, int> permutation, Dictionary<int, int> cycle,
        ref int count)
    {
```

```
        int max = cycle.ElementAt(0).Key;
        int firstCandidate = cycle.ElementAt(1).Key;
        int secondCandidate = 0;

        for (int i = 2; i < cycle.Count; i++)
        {
            secondCandidate = cycle.ElementAt(i).Key;

            if (secondCandidate > firstCandidate)
                break;

            firstCandidate = secondCandidate;
        }

        permutation.Transposition(firstCandidate,
            secondCandidate, max, ref count);
    }

    // Primenjujemo 0,2,2-potez
    private static void MakeZeroTwoTwoMove(this
        SortedDictionary<int, int> permutation, ref int count
        )
    {
        List<Dictionary<int, int>> cyclesInPermutation =
            permutation.DecomposePermutation();
        IEnumerable<Dictionary<int, int>> nonorientedCycles
            = cyclesInPermutation.Where(cycle => cycle.
            IsCycleNonoriented());
        IEnumerable<Dictionary<int, int>>
            nonorientedLongCycles = nonorientedCycles.Where(
            cycle => cycle.IsLongCycle());

        // Trazimo dva neorijentisana duga ciklusa kojima se
            trojke preplicu
        foreach (Dictionary<int, int> cycle1 in
            nonorientedLongCycles)
        {
```

```
        foreach (Dictionary<int, int> cycle2 in
            nonorientedLongCycles)
            if (!AreEquals(cycle1, cycle2) &&
                IsTripletsInCyclesInterleaving(cycle1,
                cycle2, out List<int> xyz))
            {
                // Ako smo nasli, primenjujemo 0,2,2-
                    potez
                permutation.Transposition(xyz[2], xyz
                    [1], xyz[0], ref count);
                permutation.MakeTwoMove(cycle2, ref
                    count);
                permutation.MakeTwoMove(cycle1, ref
                    count);

                return;
            }
    }

    Dictionary<int, int> cycleC = new Dictionary<int,
        int>();
    Dictionary<int, int> cycleD = new Dictionary<int,
        int>();
    Dictionary<int, int> cycleE = new Dictionary<int,
        int>();
    int s = -1;
    int i = -1;

    // Trazimo neorijentisan ciklus C koji nije
        obuhvacen drugim ciklusom
    foreach (Dictionary<int, int> cycle1 in
        nonorientedLongCycles)
    {
        bool found = true;
        foreach (Dictionary<int, int> cycle2 in
            nonorientedLongCycles)
        {
```

```
            if (!AreEquals(cycle1, cycle2) && cycle1.
                IsSpannedBy(cycle2))
            {
                found = false;
                break;
            }
        }

        if (found)
        {
            cycleC = new Dictionary<int, int>(cycle1);
            break;
        }
    }

    // Trazimo ciklus D, koji sa ciklusom C ima parove
        koji se preplicu
    foreach (Dictionary<int, int> tempCycle in
        nonorientedCycles)
    {
        if (AreEquals(cycleC, tempCycle))
            continue;

        if (HasInterleavingPair(cycleC.Last().Key,
            cycleC.First().Key, tempCycle))
        {
            cycleD = new Dictionary<int, int>(tempCycle)
                ;

            if (cycleC.Last().Key < cycleD.Last().Key)
                s = FindS(cycleC, cycleD.Last().Key);
            else
                s = FindS(cycleC, cycleD.First().Key);

            break;
        }
    }
```

```
            i = cycleC.Last().Key;

            // Trazimo ciklus E
            if (s > i)
                FindE(nonorientedCycles.ToList(), i, s, out
                    cycleE);
            else if (s == i)
            {
                int t;
                if (cycleC.Last().Key < cycleD.Last().Key)
                    t = FindT(cycleC, cycleD.Last().Key);
                else
                    t = FindT(cycleC, cycleD.First().Key);

                FindE(nonorientedCycles.ToList(), t, cycleC.
                    First().Key, out cycleE);
            }

            // Primenjujemo 0,2,2-potez
            if (cycleE.Last().Key < cycleC.Last().Key)
                permutation.Transposition(cycleD.First().Key,
                    cycleD.Last().Key, cycleE.First().Key, ref
                    count);
            else
                permutation.Transposition(cycleD.First().Key,
                    cycleD.Last().Key, cycleE.Last().Key, ref
                    count);

            permutation.MakeTwoMove(cycleC, ref count);
            permutation.MakeTwoMove(cycleD, ref count);
        }

        private static void FindE(List<Dictionary<int, int>>
            nonorientedCycles, int a, int b, out Dictionary<int,
            int> cycleE)
        {
```

```
            cycleE = new Dictionary<int, int>();
            foreach (Dictionary<int, int> tempCycle in
                nonorientedCycles)
            {
                if (HasInterleavingPair(a, b, tempCycle))
                {
                    cycleE = new Dictionary<int, int>(tempCycle)
                        ;
                    return;
                }
            }
        }

        private static int FindT(Dictionary<int, int> cycleC,
            int i)
        {
            int result = cycleC.First().Key;

            foreach (int key in cycleC.Keys)
                if (key > i && key < result)
                    result = key;

            return result;
        }

        private static int FindS(this Dictionary<int, int> cycle
            , int i)
        {
            int result = -1;

            foreach (int key in cycle.Keys)
                if (key < i && key > result)
                    result = key;

            return result;
        }
```

```
private static bool HasInterleavingPair(int a, int b,
   Dictionary<int, int> cycle2)
{
    int cycle2First = cycle2.First().Key;

    int count1 = cycle2.Count(el => el.Key < b && el.Key
        > a);
    int count2 = cycle2.Count(el => el.Key > b);
    int count3 = cycle2.Count(el => el.Key < a);

    if (count1 > 0 && (count2 > 0 ^ count3 > 0))
        return true;

    return false;
}

private static bool IsSpannedBy(this Dictionary<int, int
   > cycle1, Dictionary<int, int> cycle2)
{
    int cycle1First = cycle1.First().Key;
    int cycle1Last = cycle1.Last().Key;
    int cycle2First = cycle2.First().Key;
    int cycle2Last = cycle2.Last().Key;

    return cycle2Last < cycle1Last && cycle1First <
       cycle2First;
}

// Proveravamo da li dva ciklusa imaju trojke koje se
   preplicu
// Ulaz: ciklus1, ciklus2
// Out argument: Trojka (x,y,z) koja se preplice sa
   trojkom iz drugog ciklusa, gde je x najvece
// Izlaz: true ako su nadjene trojke koje se preplicu,
   false u suprotnom
private static bool IsTripletsInCyclesInterleaving(
   Dictionary<int, int> cycle1, Dictionary<int, int>
```

```
        cycle2, out List<int> xyz)
    {
        xyz = new List<int>();
        SortedDictionary<int, int> union = new
            SortedDictionary<int, int>();
        foreach (KeyValuePair<int, int> kvp in cycle1)
            union.Add(kvp.Key, 1);

        foreach (KeyValuePair<int, int> kvp in cycle2)
            union.Add(kvp.Key, 2);

        KeyValuePair<int, int> firstElement = union.First();
        int elementFrom = firstElement.Value;
        int counter = 1;

        SortedDictionary<int, int> candidate = new
            SortedDictionary<int, int> { { firstElement.Key,
            firstElement.Value } };

        foreach (KeyValuePair<int, int> kvp in union)
        {
            if (counter > 6)
                break;

            if (kvp.Value != elementFrom)
            {
                elementFrom = kvp.Value;
                candidate.Add(kvp.Key, kvp.Value);
                counter++;
            }
        }

        if (counter == 6)
        {
            xyz.Add(candidate.ElementAt(0).Key);
            xyz.Add(candidate.ElementAt(2).Key);
            xyz.Add(candidate.ElementAt(4).Key);
```

```
        }

        return counter == 6;
    }

    // Primenjujemo transpozicija p(i,j,k) na permutaciju
    // Ulaz: Permutacija, indeksi permutacije za koje radimo
        transpoziciju
    private static void Transposition(this SortedDictionary<
        int, int> permutation, int i, int j, int k, ref int
        count)
    {
        int n = permutation.Count;

        if (IsIndexInvalid(i, n) || IsIndexInvalid(j, n) ||
            IsIndexInvalid(k, n))
            return;

        if (i > j)
        {
            int temp = i;
            i = j;
            j = temp;
        }

        if (k < i)
        {
            int temp = i;
            i = k;
            k = j;
            j = temp;
        }

        if (k < j)
        {
            int temp = j;
            j = k;
```

```
            k = temp;
        }

        if (IsBetween(i, j, k))
            return;

        List<int> tempIndex1 = new List<int>();

        for (int y = j; y < k; y++)
        {
            permutation.TryGetValue(y, out int value);
            tempIndex1.Add(value);
            permutation.Remove(y);
        }

        for (int x = i; x < j; x++)
        {
            permutation.TryGetValue(x, out int value);
            tempIndex1.Add(value);
            permutation.Remove(x);
        }

        foreach (int el in tempIndex1)
            permutation.Add(i++, el);

        count++;
    }

    private static bool IsIndexInvalid(int i, int n)
    {
        return i < 1 || i > n - 1;
    }

    private static bool IsBetween(int i, int j, int k)
    {
        return (k >= i && k <= j) || (k <= i && k >= j);
    }
```

```
private static bool IsSorted(this SortedDictionary<int,
    int> permutation)
{
    foreach (KeyValuePair<int, int> kvp in permutation)
        if (kvp.Key != kvp.Value)
            return false;

    return true;
}

// Kreiramo listu ciklusa u permutaciji
private static List<Dictionary<int, int>>
    DecomposePermutation(this SortedDictionary<int, int>
    permutation)
{
    List<Dictionary<int, int>> result = new List<
        Dictionary<int, int>>();

    for (int i = 0; i < permutation.Count - 2; i++)
    {
        Dictionary<int, int> newCycle = permutation.
            Cycle(i);

        if (!result.Any(cycle => AreEquals(cycle,
            newCycle)))
            result.Add(newCycle);
    }

    return result;
}

// Kreiramo ciklus pocevsi od pozicije i
// Ulaz: Permutacija, indeks od kog pocinje ciklus
// Izlaz: Ciklus koji za prvi element ima najdesnju
    povratnu ivicu
```

```
private static Dictionary<int, int> Cycle(this
    SortedDictionary<int, int> permutation, int i)
{
    KeyValuePair<int, int> firstElement = permutation.
        ElementAt(i);
    KeyValuePair<int, int> backEdge = permutation.First(
        el => el.Value == firstElement.Value + 1);
    Dictionary<int, int> cycle = new Dictionary<int, int
        > { { backEdge.Key, backEdge.Value } };

    do
    {
        KeyValuePair<int, int> lastAdded = cycle.Last();

        permutation.TryGetValue(lastAdded.Key - 1, out
            int value);

        backEdge = permutation.First(el => el.Value ==
            value + 1);

        if (cycle.ContainsKey(backEdge.Key))
            break;

        cycle.Add(backEdge.Key, backEdge.Value);
    }
    while (!cycle.First().Equals(cycle.Last()));

    int rightmostIndex = cycle.Keys.Max();
    Dictionary<int, int> tempCycle1 = new Dictionary<int
        , int>();
    IEnumerable<KeyValuePair<int, int>> tempCycle2 =
        cycle.TakeWhile(el => el.Key < rightmostIndex);

    bool rightmostIndexFound = false;

    foreach (KeyValuePair<int, int> kvp in cycle)
    {
```

```
            if (kvp.Key == rightmostIndex)
                rightmostIndexFound = true;

            if (rightmostIndexFound)
                tempCycle1.Add(kvp.Key, kvp.Value);
        }

        foreach (KeyValuePair<int, int> kvp in tempCycle2)
            tempCycle1.Add(kvp.Key, kvp.Value);

        return tempCycle1;
    }

    private static bool AreEquals(Dictionary<int, int>
       cycle1, Dictionary<int, int> cycle2)
    {
        if (cycle1.Count != cycle2.Count)
            return false;

        return cycle2.ContainsKey(cycle1.First().Key);
    }

    private static bool IsLongCycle(this Dictionary<int, int
       > cycle)
    {
        return cycle.Count > 2;
    }

    private static bool HasLongCycle(this SortedDictionary<
       int, int> permutation)
    {
        return permutation.DecomposePermutation().Count(el
           => el.IsLongCycle()) > 0;
    }

    private static bool GetTwoCycle(this SortedDictionary<
       int, int> permutation, out Dictionary<int, int> cycle
```

```
    )
{
    cycle = permutation.DecomposePermutation().
        FirstOrDefault(el => el.IsTwoCycle());

    if (cycle != null)
        return true;

    return false;
}

private static bool GetLongOrientedCycle(this
    SortedDictionary<int, int> permutation, out
    Dictionary<int, int> cycle)
{
    cycle = permutation.DecomposePermutation().
        FirstOrDefault(el => el.IsLongCycle() && !el.
        IsCycleNonoriented());

    if (cycle != null)
        return true;

    return false;
}

private static bool IsTwoCycle(this Dictionary<int, int>
    cycle)
{
    return cycle.Count == 2;
}

private static bool IsCycleNonoriented(this Dictionary<
    int, int> cycle)
{
    for (int i = 0; i < cycle.Count - 1; i++)
        if (cycle.ElementAt(i).Key < cycle.ElementAt(i +
            1).Key)
```

```
                    return false;

            return true;
        }
    }
}
```

Алгоритам *TransSort*

Имплементација алгоритма *TransSort* описаног у одељку 2.5. Пример покретања програма налази се у одељку 3.1, а резултати израчунавања су приказани у табели 3.1 у одељку 3.2.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace SortingByTranspositions
{
    internal static class TransSortImplementation
    {
        // Sortiranje permutacije Bafna-Pevzner algoritmom
           koriscenjem validnih 0-poteza i 2-poteza
        // Ulaz: Pocetna permutacija
        // Out argument: Rastojanje pocetne permutacije do
           identicke
        // Izlaz: Sortirana permutacija
        public static void TransSort(this SortedDictionary<int,
           int> permutation, out int count)
        {
            count = 0;
            while (!permutation.IsSorted())
            {
                // Moguce je da orijentisan dug ciklus ima
                   validan 2-potez ili validan 0,2,2-potez
                if (permutation.GetLongOrientedCycle(out
                   Dictionary<int, int> longOrientedCycle))
                    permutation.MakeValidMoves(longOrientedCycle
                       , ref count);
                // Primenjujemo validan 0,2,2-potez
                else if (permutation.HasLongCycle())
                    permutation.MakeValidZeroTwoTwoMove(ref
                       count);
                else if (permutation.GetTwoCycle(out Dictionary<
                   int, int> twoCycle))
```

```
            {
                // Primenjujemo validan 2-potez
                if (!twoCycle.IsCycleNonoriented())
                    permutation.MakeValidTwoMove(twoCycle,
                        ref count);
                // Primenjujemo dobar 0-potez i validan 2-
                    potez
                else
                    permutation.MakeGoodZeroValidTwoMove(
                        twoCycle, ref count);
            }
        }
    }

    // Proveravamo da li je moguce primeniti validan 2-potez
        na ciklusu i ako jeste primenjujemo ga, ako nije
    primenjujemo validan 0,2,2-potez
    // Ulaz: Permutacija i orijentisan dug ciklus
    // Out argument: Broj permutacije koje smo primenili (1
    - za validan 2-potez i 3 za validan 0,2,2-potez
    private static void MakeValidMoves(this SortedDictionary
        <int, int> permutation, Dictionary<int, int> cycle,
        ref int count)
    {
        if (!permutation.TryMakeValidTwoMove(cycle, ref
            count, out int x, out int y, out int z, out int a
            , out int b))
        {
            permutation.Transposition(b, z, x, ref count);
            permutation.Transposition(b, y, x, ref count);
            permutation.Transposition(a, z, x, ref count);
            return;
        }
    }

    private static bool TryMakeValidTwoMove(this
        SortedDictionary<int, int> permutation, Dictionary<
```

```
        int, int> cycle, ref int count, out int x, out int y,
         out int z, out int a, out int b)
    {
         // U ciklusu nalazimo Skups S, tako da vazi S = {(x,
             y, z) | d(x, y) = 1}, gde je d(x, y) broj
            povratnih ivica izmedju x i y u ciklusu
         HashSet<List<int>> setS = cycle.FindSetS();
         int maxX = 0;
         x = 0;
         y = 0;
         z = 0;
         a = 0;
         b = 0;

         if (setS.Count == 0)
             return false;

         // Nalazimo trojku iz S, tako da je x najvece
         foreach (List<int> triple in setS)
             if (triple[0] > maxX)
                 maxX = triple[0];

         List<int> tripleXYZ = setS.First(triple => triple[0]
             == maxX);

         x = tripleXYZ[0];
         y = tripleXYZ[1];
         z = tripleXYZ[2];

         // Ako nam je d(z, x) ili d(y, z) neparno mozemo
            primeniti validan 2-potez, jer je d(x, y) = 1
         if (cycle.EdgeDistance(x, y) % 2 == 1 && (cycle.
            EdgeDistance(z, x) % 2 == 1 || cycle.EdgeDistance
            (y, z) % 2 == 1))
         {
             permutation.Transposition(y, z, x, ref count);
             return true;
```

```
        }

        // Nalazimo povratne ivice koje se nalaze pre i
           posle z u ciklusu
        for (int i = 0; i < cycle.Count; i++)
        {
            if (cycle.ElementAt(i).Key == z)
            {
                a = cycle.ElementAt(i - 1).Key;
                b = cycle.ElementAt(i + 1).Key;
                break;
            }
        }

        // Ispitujemo da li vaze uslovi za primenu validnog
           2-poteza
        if (y < a && a < x)
        {
            permutation.Transposition(y, a, x, ref count);
            return true;
        }
        else if (y < b && b < x)
        {
            permutation.Transposition(y, b, x, ref count);
            return true;
        }

        if (a < y && b < y && b < a)
        {
            permutation.Transposition(b, a, z, ref count);
            return true;
        }

        return false;
    }
```

```
private static int IndexOfEdge(this Dictionary<int, int>
    cycle, int i)
{
    for (int j = 0; j < cycle.Count; j++)
    {
        if (cycle.ElementAt(j).Key == i)
            return j;
    }

    return -1;
}

private static int EdgeDistance(this Dictionary<int, int
   > cycle, int i, int j)
{
    int indexI = IndexOfEdge(cycle, i);
    int indexJ = IndexOfEdge(cycle, j);

    if (indexI == -1 || indexJ == -1)
        return -1;

    return Math.Abs(indexI - indexJ);
}

// Primenjujemo dobar 0-potez i validan 2-potez na 2-
   ciklus u permutaciji
private static void MakeGoodZeroValidTwoMove(this
   SortedDictionary<int, int> permutation, Dictionary<
   int, int> twoCycle, ref int count)
{
    int a = twoCycle.ElementAt(1).Key;
    int b = twoCycle.ElementAt(0).Key;
    int max = 0;
    int r = 0;

    for (int i = a; i < b; i++)
    {
```

```
            KeyValuePair<int, int> element = permutation.
                ElementAt(i);
            if (element.Value > max)
            {
                max = element.Value;
                r = element.Key;
            }
        }

        permutation.TryGetValue(r, out int pi_r);
        int s = permutation.First(el => el.Value == pi_r +
            1).Key;

        if (s >= a && s <= b)
            return;

        if (s > b)
            permutation.Transposition(r + 1, s, a, ref count
                );
        else
            permutation.Transposition(a, b, s, ref count);

        List<Dictionary<int, int>> cycles = permutation.
            DecomposePermutation();
        Dictionary<int, int> cycleForTwoMove = cycles.First(
            cycle => cycle.ContainsKey(a));

        permutation.MakeValidTwoMove(cycleForTwoMove, ref
            count);
    }

    private static bool GetLongOrientedCycle(this
        SortedDictionary<int, int> permutation, out
        Dictionary<int, int> cycle)
    {
        cycle = permutation.DecomposePermutation().
            FirstOrDefault(el => el.IsLongCycle() && !el.
```

```
            IsCycleNonoriented());

        if (cycle != null)
            return true;

        return false;
    }

    // Primeni validan 2-potez na ciklus
    // Ulaz: permutacija i ciklus na koji deluje
       transpozicija
    private static void MakeValidTwoMove(this
       SortedDictionary<int, int> permutation, Dictionary<
       int, int> cycle, ref int count)
    {
        int max = cycle.ElementAt(0).Key;
        int firstCandidate = cycle.ElementAt(1).Key;
        int secondCandidate = 0;

        for (int i = 2; i < cycle.Count; i++)
        {
            secondCandidate = cycle.ElementAt(i).Key;

            if (secondCandidate > firstCandidate)
                break;

            firstCandidate = secondCandidate;
        }

        permutation.Transposition(firstCandidate,
           secondCandidate, max, ref count);
    }

    private static HashSet<List<int>> FindSetS(this
       Dictionary<int, int> cycle)
    {
```

```
        HashSet<List<int>> result = new HashSet<List<int>>()
            ;
        for (int i = 0; i < cycle.Count - 2; i++)
            for (int j = i + 1; j < cycle.Count - 1; j++)
                for (int k = j + 1; k < cycle.Count; k++)
                {
                    if (cycle.ElementAt(i).Key > cycle.
                        ElementAt(j).Key &&
                        cycle.ElementAt(j).Key > cycle.
                            ElementAt(k).Key)
                        continue;

                    if (cycle.EdgeDistance(cycle.ElementAt(i
                        ).Key, cycle.ElementAt(j).Key) == 1)
                        result.Add(new List<int> { cycle.
                            ElementAt(i).Key, cycle.ElementAt
                            (j).Key, cycle.ElementAt(k).Key
                            });
                }

        return result;
    }

    //Na permutaciju primenimo validan 0,2,2-potez
    private static void MakeValidZeroTwoTwoMove(this
        SortedDictionary<int, int> permutation, ref int count
        )
    {
        List<Dictionary<int, int>> cyclesInPermutation =
            permutation.DecomposePermutation();
        IEnumerable<Dictionary<int, int>> nonorientedCycles
            = cyclesInPermutation.Where(cycle => cycle.
            IsCycleNonoriented());
        IEnumerable<Dictionary<int, int>>
            nonorientedLongCycles = nonorientedCycles.Where(
            cycle => cycle.IsLongCycle());
```

```
// Trazimo dva neorijentisana ciklusa kojima se
   trojke preplicu
foreach (Dictionary<int, int> cycle1 in
   nonorientedLongCycles)
{
    foreach (Dictionary<int, int> cycle2 in
       nonorientedLongCycles)
        if (!AreEquals(cycle1, cycle2) &&
           IsTripletsInCyclesInterleaving(cycle1,
           cycle2, out List<int> xyz))
        {
            // Nasli smo takva dva ciklusa i
               primenjujemo validan 0,2,2-potez
            permutation.Transposition(xyz[2], xyz
               [1], xyz[0], ref count);
            permutation.TryMakeValidTwoMove(cycle1,
               ref count, out _, out _, out _, out _
               , out _);
            permutation.TryMakeValidTwoMove(cycle2,
               ref count, out _, out _, out _, out _
               , out _);
            return;
        }
}

Dictionary<int, int> cycleC = new Dictionary<int,
   int>();
Dictionary<int, int> cycleD = new Dictionary<int,
   int>();
Dictionary<int, int> cycleE = new Dictionary<int,
   int>();
int s = -1;
int i = -1;

// Trazimo neorijentisan ciklus C koji nije
   obuhvacen ni jednim drugim ciklusom
```

```
foreach (Dictionary<int, int> cycle1 in
    nonorientedLongCycles)
{
     bool found = true;
     foreach (Dictionary<int, int> cycle2 in
         nonorientedLongCycles)
     {
          if (!AreEquals(cycle1, cycle2) && cycle1.
              IsSpannedBy(cycle2))
          {
               found = false;
               break;
          }
     }

     if (found)
     {
          cycleC = new Dictionary<int, int>(cycle1);
          break;
     }
}

// Trazimo ciklus D, koji sa ciklusom C ima parove
    koji se preplicu
foreach (Dictionary<int, int> tempCycle in
    nonorientedCycles)
{
     if (AreEquals(cycleC, tempCycle))
          continue;

     if (HasInterleavingPair(cycleC.Last().Key,
         cycleC.First().Key, tempCycle))
     {
          cycleD = new Dictionary<int, int>(tempCycle)
              ;
          break;
     }
```

```
}

if (cycleC.Last().Key < cycleD.Last().Key)
    s = FindS(cycleC, cycleD.Last().Key);
else
    s = FindS(cycleC, cycleD.First().Key);

i = cycleC.Last().Key;

// Trazimo ciklus E
if (s > i)
    FindE(nonorientedCycles.ToList(), i, s, out
       cycleE);
else if (s == i)
{
    int t;
    if (cycleC.Last().Key < cycleD.Last().Key)
        t = FindT(cycleC, cycleD.Last().Key);
    else
        t = FindT(cycleC, cycleD.First().Key);

    FindE(nonorientedCycles.ToList(), t, cycleC.
       First().Key, out cycleE);
}

// Primenjujemo validan 0,2,2-potez
if (cycleE.Last().Key < cycleC.Last().Key)
    permutation.Transposition(cycleD.First().Key,
       cycleD.Last().Key, cycleE.First().Key, ref
       count);
else
    permutation.Transposition(cycleD.First().Key,
       cycleD.Last().Key, cycleE.Last().Key, ref
       count);

permutation.TryMakeValidTwoMove(permutation.Cycle(
   cycleC.First().Key - 1), ref count, out _, out _,
```

```
            out _, out _, out _);
        permutation.TryMakeValidTwoMove(permutation.Cycle(
            cycleD.First().Key - 1), ref count, out _, out _,
            out _, out _, out _);
        return;
    }

    private static void FindE(List<Dictionary<int, int>>
        nonorientedCycles, int a, int b, out Dictionary<int,
        int> cycleE)
    {
        cycleE = new Dictionary<int, int>();
        foreach (Dictionary<int, int> tempCycle in
            nonorientedCycles)
        {
            if (HasInterleavingPair(a, b, tempCycle))
            {
                cycleE = new Dictionary<int, int>(tempCycle)
                    ;
                return;
            }
        }
    }

    private static int FindT(Dictionary<int, int> cycle, int
        i)
    {
        int result = cycle.First().Key;

        foreach (int key in cycle.Keys)
            if (key > i && key < result)
                result = key;

        return result;
    }
```

```
private static int FindS(this Dictionary<int, int> cycle
    , int i)
{
    int result = 0;

    foreach (int key in cycle.Keys)
        if (key < i && key > result)
            result = key;

    return result;
}

private static bool HasInterleavingPair(int a, int b,
    Dictionary<int, int> cycle2)
{
    int count1 = cycle2.Count(el => el.Key > a && el.Key
        < b);
    int count2 = cycle2.Count(el => el.Key < a);
    int count3 = cycle2.Count(el => el.Key > b);

    if (count1 > 0 && (count2 > 0 ^ count3 > 0))
        return true;

    return false;
}

private static bool IsSpannedBy(this Dictionary<int, int
    > cycle1, Dictionary<int, int> cycle2)
{
    int cycle1First = cycle1.First().Key;
    int cycle1Last = cycle1.Last().Key;
    int cycle2First = cycle2.First().Key;
    int cycle2Last = cycle2.Last().Key;

    return cycle2Last < cycle1Last && cycle1First <
        cycle2First;
}
```

```
// Proveravamo da li dva ciklusa imaju trojke koje se
   preplicu
// Ulaz: ciklus1, ciklus2
// Out argument: Trojka (x,y,z) koja se preplice sa
   trojkom iz drugog ciklusa, gde je x najvece
// Izlaz: true ako su nadjene trojke koje se preplicu,
   false u suprotnom
private static bool IsTripletsInCyclesInterleaving(
   Dictionary<int, int> cycle1, Dictionary<int, int>
   cycle2, out List<int> xyz)
{
    xyz = new List<int>();
    SortedDictionary<int, int> union = new
       SortedDictionary<int, int>();
    foreach (KeyValuePair<int, int> kvp in cycle1)
        union.Add(kvp.Key, 1);

    foreach (KeyValuePair<int, int> kvp in cycle2)
        union.Add(kvp.Key, 2);

    KeyValuePair<int, int> firstElement = union.First();
    int elementFrom = firstElement.Value;
    int counter = 1;

    SortedDictionary<int, int> candidate = new
       SortedDictionary<int, int> { { firstElement.Key,
       firstElement.Value } };

    foreach (KeyValuePair<int, int> kvp in union)
    {
        if (counter > 6)
            break;

        if (kvp.Value != elementFrom)
        {
            elementFrom = kvp.Value;
```

```
                candidate.Add(kvp.Key, kvp.Value);
                counter++;
            }
        }

        if (counter == 6)
        {
            for (int i = 0; i < candidate.Count; i++)
            {
                int candidateKey = candidate.ElementAt(i).
                   Key;
                if (cycle2.ContainsKey(candidateKey))
                    xyz.Add(candidateKey);
            }
        }

        return counter == 6;
    }

    // Primenjujemo transpozicija p(i,j,k) na permutaciju
    // Ulaz: Permutacija, indeksi permutacije za koje radimo
        transpoziciju
    private static void Transposition(this SortedDictionary<
       int, int> permutation, int i, int j, int k, ref int
       count)
    {
        int n = permutation.Count;
        if (IsIndexInvalid(i, n) || IsIndexInvalid(j, n) ||
           IsIndexInvalid(k, n))
            return;

        if (i > j)
        {
            int temp = i;
            i = j;
            j = temp;
        }
```

```
if (k < i)
{
    int temp = i;
    i = k;
    k = j;
    j = temp;
}

if (k < j)
{
    int temp = j;
    j = k;
    k = temp;
}

if (IsBetween(i, j, k))
    return;

List<int> tempIndex1 = new List<int>();

for (int y = j; y < k; y++)
{
    permutation.TryGetValue(y, out int value);
    tempIndex1.Add(value);
    permutation.Remove(y);
}

for (int x = i; x < j; x++)
{
    permutation.TryGetValue(x, out int value);
    tempIndex1.Add(value);
    permutation.Remove(x);
}

foreach (int el in tempIndex1)
    permutation.Add(i++, el);
```

```
        count++;
    }

    private static bool IsIndexInvalid(int i, int n)
    {
        return i < 1 || i > n - 1;
    }

    private static bool IsBetween(int i, int j, int k)
    {
        return (k >= i && k <= j) || (k <= i && k >= j);
    }

    private static bool IsSorted(this SortedDictionary<int,
       int> permutation)
    {
        foreach (KeyValuePair<int, int> kvp in permutation)
            if (kvp.Key != kvp.Value)
                return false;

        return true;
    }

    // Kreiramo listu ciklusa u permutaciji
    private static List<Dictionary<int, int>>
       DecomposePermutation(this SortedDictionary<int, int>
       permutation)
    {
        List<Dictionary<int, int>> result = new List<
           Dictionary<int, int>>();

        for (int i = 0; i < permutation.Count - 2; i++)
        {
            Dictionary<int, int> newCycle = permutation.
               Cycle(i);
```

```
            if (!result.Any(cycle => AreEquals(cycle,
                newCycle)))
                result.Add(newCycle);
        }

        return result;
    }

    // Kreiramo ciklus pocevsi od pozicije i
    // Ulaz: Permutacija, indeks od kog pocinje ciklus
    // Izlaz: Ciklus koji za prvi element ima najdesnju
       povratnu ivicu
    private static Dictionary<int, int> Cycle(this
       SortedDictionary<int, int> permutation, int i)
    {
        KeyValuePair<int, int> firstElement = permutation.
           ElementAt(i);
        KeyValuePair<int, int> backEdge = permutation.First(
           el => el.Value == firstElement.Value + 1);
        Dictionary<int, int> cycle = new Dictionary<int, int
           > { { backEdge.Key, backEdge.Value } };

        do
        {
            KeyValuePair<int, int> lastAdded = cycle.Last();

            permutation.TryGetValue(lastAdded.Key - 1, out
               int value);

            backEdge = permutation.First(el => el.Value ==
               value + 1);

            if (cycle.ContainsKey(backEdge.Key))
                break;

            cycle.Add(backEdge.Key, backEdge.Value);
        }
```

```
        while (!cycle.First().Equals(cycle.Last()));

        int rightmostIndex = cycle.Keys.Max();
        Dictionary<int, int> tempCycle1 = new Dictionary<int
            , int>();
        IEnumerable<KeyValuePair<int, int>> tempCycle2 =
           cycle.TakeWhile(el => el.Key < rightmostIndex);

        bool rightmostIndexFound = false;

        foreach (KeyValuePair<int, int> kvp in cycle)
        {
            if (kvp.Key == rightmostIndex)
                rightmostIndexFound = true;

            if (rightmostIndexFound)
                tempCycle1.Add(kvp.Key, kvp.Value);
        }

        foreach (KeyValuePair<int, int> kvp in tempCycle2)
            tempCycle1.Add(kvp.Key, kvp.Value);

        return tempCycle1;
    }

    private static bool AreEquals(Dictionary<int, int>
       cycle1, Dictionary<int, int> cycle2)
    {
        if (cycle1.Count != cycle2.Count)
            return false;

        return cycle2.ContainsKey(cycle1.First().Key);
    }

    private static bool IsLongCycle(this Dictionary<int, int
       > cycle)
    {
```

```
        return cycle.Count > 2;
    }

    private static bool IsTwoCycle(this Dictionary<int, int>
        cycle)
    {
        return cycle.Count == 2;
    }

    private static bool IsCycleNonoriented(this Dictionary<
       int, int> cycle)
    {
        for (int i = 0; i < cycle.Count - 1; i++)
            if (cycle.ElementAt(i).Key < cycle.ElementAt(i +
                1).Key)
                return false;

        return true;
    }

    private static bool GetTwoCycle(this SortedDictionary<
       int, int> permutation, out Dictionary<int, int> cycle
       )
    {
        cycle = permutation.DecomposePermutation().
           FirstOrDefault(el => el.IsTwoCycle());

        if (cycle != null)
            return true;

        return false;
    }

    private static bool HasLongCycle(this SortedDictionary<
       int, int> permutation)
    {
```

```
            return permutation.DecomposePermutation().Count(el
                => el.IsLongCycle()) > 0;
        }

        /// <summary>
        /// Broj neparnih ciklusa u permutaciji.
        /// </summary>
        public static int NumberOfOddCycles(this
            SortedDictionary<int, int> permutation)
        {
            return permutation.DecomposePermutation().Count(
                cycle => cycle.Count % 2 == 1);
        }
    }
}
```