ალგორითმების აგება

ლელა ალხაზიშვილი
ალექსანდრე გამყრელიძე

ნახაზები, მაგალითები, დავალებები და LATEX: ლევან კასრაძე

სარჩევი

1 ძირითადი ცნებები გრაფთა თეორიიდან 5
- 1.1 შესავალი 5
- 1.2 ძირითადი ცნებები და განსაზღვრებები გრაფებზე 7
- 1.3 ხეები 10
- 1.4 გრაფის წარმოდგენა 12
- 1.5 სავარჯიშოები 13

2 გრაფის შემოვლის ალგორითმები 17
- 2.1 სიგანეში ძებნის ალგორითმი 17
- 2.2 სიღრმეში ძებნის ალგორითმი 21
- 2.3 წიბოთა კლასიფიკაცია 28
- 2.4 ტოპოლოგიური სორტირება 28
- 2.5 ძლიერად ბმული კომპონენტები 30
- 2.6 სავარჯიშოები 32

3 მინიმალური დამფარავი ხეები 35
- 3.1 კრასკალის ალგორითმი 37
- 3.2 პრიმის ალგორითმი 40
- 3.3 სავარჯიშოები 42

4 უმოკლესი გზები ერთი წვეროდან 45
- 4.1 უმოკლესი გზის პოვნის ამოცანა 46
- 4.2 ბელმან-ფორდის ალგორითმი 50
- 4.3 უმოკლესი გზები აციკლურ ორიენტირებულ გრაფში 52
- 4.4 დეიქსტრას ალგორითმი 53
- 4.5 იენის ალგორითმი 55
- 4.6 სავარჯიშოები 56

5 უმოკლესი გზები წვეროთა ყველა წყვილისათვის 57
- 5.1 ფლოიდ-ვორშელის ალგორითმი 57
- 5.2 ორიენტირებული გრაფის ტრანზიტული ჩაკეტვა 59
- 5.3 ჯონსონის ალგორითმი ხალვათი გრაფებისათვის 62
- 5.4 სავარჯიშოები 64

6 ამოცანათა გრაფებზე გადატანის მაგალითები 65
- 6.1 მოძრაობა ვიდეო თამაშებში. 65
- 6.2 გენეტური კოდის აგება. 66
- 6.3 ობიექტების დაჯგუფება. 66
- 6.4 სიტყვათა შემოკლება 67
- 6.5 გაყალბების აღმოჩენა. 67
- 6.6 ეკონომიკური ამოცანები 68

7 არითმეტიკული ალგორითმები და მათი გამოყენება 71
7.1 ბულის ალგებრის ელემენტები . . . 71
7.2 n ბიტიანი რიცხვების მიმატება . . . 75
7.3 n ბიტიანი რიცხვების გამოკლება . . . 80
7.4 n ბიტიანი რიცხვების გამრავლება . . . 81

8 დინამიკური პროგრამირება 85
8.1 ფიბონაჩის რიცხვების მოძებნა . . . 85
8.2 დინამიკური პროგრამირების ამოცანების სპეციფიკა . . . 85
8.3 უგრძესი გზის პოვნა რიცხვების სამკუთხა ცხრილში . . . 86
8.4 უდიდესი საერთო ქვემიმდევრობის პოვნა . . . 88
8.5 მატრიცათა მიმდევრობის გადამრავლების ამოცანა . . . 90
8.6 მრავალკუთხედის ოპტიმალური ტრიანგულაცია . . . 95
8.7 სავარჯიშოები . . . 96

9 ხარბი ალგორითმები 97
9.1 ამოცანა განაცხადების შერჩევაზე . . . 97
9.2 როდის გამოვიყენოთ ხარბი ალგორითმი? . . . 99
9.3 მატროიდები . . . 100
9.4 ხარბი ალგორითმები აწონილი მატროიდისთვის . . . 102
9.5 სავარჯიშოები . . . 105

10 ქვესტრიქონების ძებნის ამოცანა 107
10.1 აღნიშვნები და ტერმინოლოგია . . . 107
10.2 ქვესტრიქონების ძებნის ამოცანის დასმა . . . 107
10.3 ქვესტრიქონების ძებნის უმარტივესი ალგორითმი . . . 108
10.4 კნუტ-მორის-პრატის ალგორითმი . . . 109
10.5 ბოიერ-მურ-ჰორსპულის ალგორითმი . . . 112
10.6 ბოიერ-მურის ალგორითმი . . . 114
10.7 სავარჯიშოები . . . 117

თავი 1

ძირითადი ცნებები გრაფთა თეორიიდან

1.1 შესავალი

კომპიუტერული მეცნიერების სპეციალისტისთვის ალგორითმების შესწავლა მნიშვნელოვანია როგორც პრაქტიკული, ისე თეორიული თვალსაზრისით. პრაქტიკულად მას უნდა ჰქონდეს წარმოდგენა ძირითად ალგორითმებზე, რომლებიც შესაბამის მონაცემთა სტრუქტურის არჩევით, პროგრამული ინსტრუმენტების გამოყენებით, მისცემს ალგორითმის ეფექტურად იმპლემენტაციის საშუალებას ამა თუ იმ ამოცანისთვის. თეორიული თვალსაზრისით, ალგორითმები წარმოადგენენ ბაზისს, რომლის გარეშეც შეუძლებელია პროგრამული კოდის დაწერა.
ალგორითმების შესწავლა არ გულისხმობს რამდენიმე ალგორითმის აღწერას რამდენიმე კონკრეტული ამოცანისთვის, არამედ სტილის და მიდგომების გააზრებას, რომელიც გამოადგება პროგრამისტს ახალი ამოცანის დასმისას, ალგორითმის გამოყენებაში. მეორე მხრივ, პროგრამისტმა უნდა შეძლოს გაარჩიოს ამოხსნადია თუ არა დასმული ამოცანა.
არსებობს ალგორითმების კლასიფიკაციის ორი მიდგომა: ამოცანის ტიპის და პროექტირების მეთოდის მიხედვით.
ამოცანის ტიპის მიხედვით ალგორითმები შეიძლება დაიყოს შემდეგნაირად:

- დახარისხება
- ძებნა
- სტრიქონის დამუშავება
- ამოცანები გრაფებზე
- კომბინატორული ამოცანები
- გეომეტრიული ამოცანები
- რიცხვითი ამოცანები

პროექტირების მეთოდის მიხედვით ალგორითმები შეიძლება დაიყოს:

- "უხეში ძალის" მეთოდი
- დეკომპოზიციის მეთოდი
- ამოცანის ზომის შემცირების მეთოდი
- გარდაქმნის მეთოდი
- დროისა და სივრცის კომპრომისის მეთოდი
- ხარბი მეთოდი
- დინამიკური დაპროგრამების მეთოდი
- დაბრუნებით ძებნის მეთოდი
- შტოებისა და საზღვრების მეთოდი

ჩვენს მიერ შესასწავლი ალგორითმების უმრავლესობას აქვს პოლინომიალური მუშაობის დრო, რაც ნიშნავს, რომ უარეს შემთხვევაში მუშაობის დრო არის $O(n^k)$. ასეთ ამოცანებს მიაკუთვნებენ P კლასს. უფრო ფორმალურად, P-ში შედის მხოლოდ **გადაწყვეტილების მიღების** ამოცანები (decision problems), ანუ ამოცანები, რომელზეც პასუხია "კი" ან "არა". მაგრამ ყველა ამოცანის ამოხსნა არ შეიძლება პოლინომიალურ დროში. არსებობს **გადაწყვეტილების მიღების** ამოცანები, რომელიც არ იხსნება საერთოდ, არც ერთი ალგორითმით. ასეთ ამოცანებს უწოდებენ **ამოუხსნადს** (undecidable). ამ ტიპის ამოცანის ცნობილი მაგალითია ალან ტიურინგის მიერ 1936 წელს დასმული გაჩერების ამოცანა (halting problem): მოცემული კომპიუტერული პროგრამისთვის და შემავალი მონაცემებისთვის, განსაზღვრეთ დაამთავრებს თუ არა პროგრამა მუშაობას, თუ ის იმუშავებს უსასრულოდ.
NP კლასში შედის გადაწყვეტილების მიღების ამოცანები, რომლებიც "ექვემდებარებიან შემოწმებას" პოლინომიალურ დროში. სხვა სიტყვებით რომ ვთქვათ, თუ გვაქვს ამ ამოცანის რაიმე ამონახსნი, პოლინომიალურ დროში შეიძლება შევამოწმოთ მისი კორექტულობა. P კლასის ნებისმიერი ამოცანა ეკუთვნის NP კლასს: $P \subseteq NP$. მაგრამ ჯერჯერობით (კუკის მიერ, მისი დასმის მომენტიდან, 1971 წ.) ღიად რჩება საკითხი: P კლასი NP კლასის მკაცრი ქვესიმრავლეა, თუ ეს ორი კლასი ემთხვევა ერთმანეთს?
ამოცანებისთვის, ამოცანათა კლასიდან, რომელიც ცნობილია NP-სრული კლასის სახელწოდებით, არ არის ნაპოვნი ამოხსნის ეფექტური ალგორითმები, მაგრამ ისიც არ არის დამტკიცებული, რომ ასეთი ალგორითმები არ არსებობს. მეორე მხრივ, NP-სრულ ამოცანებს აქვთ ერთი შესანიშნავი თვისება: თუ ეფექტური ალგორითმი არსებობს ერთი მაინც ამოცანისთვის ამ კლასიდან, მაშინ მისი ფორმულირება შეიძლება ამ კლასის ყველა დანარჩენი ამოცანისთვისაც.
NP-სრული ამოცანების განსაკუთრებული თვისება არის ის, რომ ზოგიერთი მათგანი, ერთი შეხედვით, ანალოგიურია ამოცანებისა, რომელთათვისაც არსებობს ალგორითმები პოლინომიალური მუშაობის დროით. მაგალითად, განვიხილოთ წყვილები, რომლებშიც ერთი იხსნება პოლინომიალურ დროში, ხოლო მეორე - NP-სრული ამოცანაა.

1. ეილერის და ჰამილტონის ციკლების:

   ეილერის ციკლის პოვნის ამოცანა: ვიპოვოთ G=(V,E) ბმულ ორიენტირებულ გრაფში ციკლი, რომელიც ყველა წიბოს შემოივლის მხოლოდ ერთხელ, თუმცა დასაშვებია წვეროების რამდენჯერმე შემოვლა. ეილერის ციკლის არსებობის დადგენა, აგრეთვე, მისი შემადგენელი წიბოების პოვნა შეიძლება O(E) დროში.

   ჰამილტონის ციკლის პოვნის ამოცანა: ვიპოვოთ G=(V,E) ორიენტირებულ გრაფში მარტივი ციკლი, რომელიც ყველა წვეროს შემოივლის. ჰამილტონის ციკლის არსებობის დადგენის ამოცანა არის NP-სრული.

2. გრაფში ორ წვეროს შორის უმოკლესი გზის და ყველაზე გრძელი გზის პოვნის ამოცანები:

   G=(V,E) ორიენტირებულ გრაფში ორ წვეროს შორის უმოკლესი გზის პოვნის ამოცანა შეიძლება ამოიხსნას O(VE) დროში. ორ წვეროს შორის ყველაზე გრძელი გზის პოვნის ამოცანა რთულია. ამოცანა იმის შესახებ, შეიცავს თუ არა გრაფი მარტივ გზას, რომელშიც წიბოების რაოდენობა მოცემულ რიცხვზე ნაკლები არაა, NP-სრულია.

   თუ რაიმე ამოცანისთვის დადგინდა, რომ იგი NP-სრულია, მაშინ პროგრამისტი უფრო ეფექტურად დახარჯავს დროს, თუ შექმნის ამ ამოცანისთვის მიახლოებით ალგორითმს ან ამოხსნის ამ ამოცანის მარტივ ვარიანტს, იმის მაგივრად, რომ ეძებოს სწრაფი ალგორითმი, რომელიც იძლევა ზუსტ ამონახსნს.

ალგორითმების აგების ამ კურსს ვიწყებთ გრაფებზე ალგორითმების შესწავლით, ამისთვის გავეცნოთ ძირითად ცნებებს და განსაზღვრებებს გრაფების შესახებ.

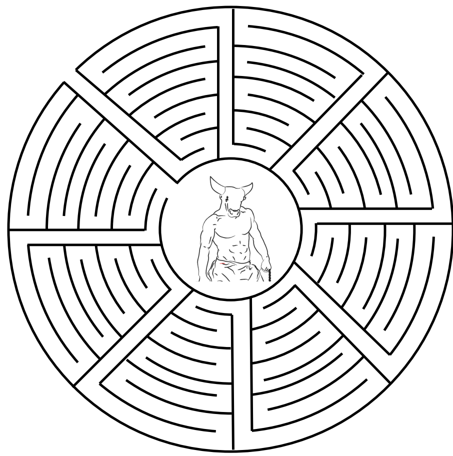
1.2 ძირითადი ცნებები და განსაზღვრებები გრაფებზე

როგორც წინა სემესტრის შესავალ კურსში აღვნიშნეთ, გრაფებით *ძალიან ბევრი* პრობლემის აღწერა და გადაჭრა შეიძლება. თუ მოცემულია რაიმე ამოცანა A, მისი მონაცემების გარდაქმნა შეიძლება ისეთ გრაფად, რომელზედაც რაღაცა სხვა ამოცანის ამოხსნით ამ საწყისი პრობლემის პასუხის დადგენა იქნება შესაძლებელი.
მაგალითად, მანქანებში ჩადგმული ნავიგაციის სისტემები, რომლელთა მეშვეობითაც ქალაქის ერთი ადგილიდან მეორეზე მისვლის უმოკლეს გზას ვგებულობთ, გრაფებზე ორ წვეროს შორის უმოკლესი გზის პოვნაზე დაიყვანება: თუ ქუჩების გადაკვეთას აღვნიშნავთ როგორც გრაფის წვეროებს, ხოლო წიბოებით კი თვითონ ქუჩებს, გამოგვივა შეწონილი გრაფი, რომელშიც ქუჩების სიგრძე წიბოს წონის ტოლი იქნება. ცხადია, რომ გრაფში უმოკლესი გზის პოვნა ქალაქში უმოკლესი გზის პოვნის ტოლფასი იქნება.
პირველ რიგში, აუცილებელია გადასაჭრელი ამოცანის მკაფიოდ და *სწორად* გადატანა გრაფებზე, რის შემდეგაც მის ამოსახსნელად რამოდენიმე ფუნდამენტური ალგორითმის ცოდნაა საჭირო, მათ შორის (მაგრამ არა მხოლოდ) უმცირესი დამფარავი ხის, მოცემული ორი წვეროს შორის უმოკლესი გზის, გრაფის პლანარულად (ბრტყლად) სიბრტყეზე დახაზვის ამოცანები. თუ ადამიანი რამოდენიმე ძირითადი ამოცანის გადაჭრის ხერხს დაეუფლება, უფრო რთული ამოცანების გადაჭრა, როგორც წესი, ამ ძირითადი ამოცანების თანმიმდევრულად გადაჭრის საშუალებითაც შეიძლება.
ამოცანათა გადაჭრის ძირითად მეთოდებს შორის (რომელიც ფართოდ გამოიყენება გრაფებზე ალგორითმებში) შეიძლება მოვიყვანოთ ე.წ. „სიგანეში ძებნის" და „სიღრმეში ძებნის" ალგორითმები, რომელთა საშუალებითაც სწრაფად შეგვიძლია შემოვიაროთ გრაფის ყველა წვერო (ყოველგვარი შეზღუდვის გარეშე).

ალგორითმებისა და მათი გადაჭრის მეთოდების შესწავლა უმჯობესია პრაქტიკული პრობლემების განხილვით დავიწყოთ.

1.2.1 ბერძნული მითი მინოტავრის შესახებ

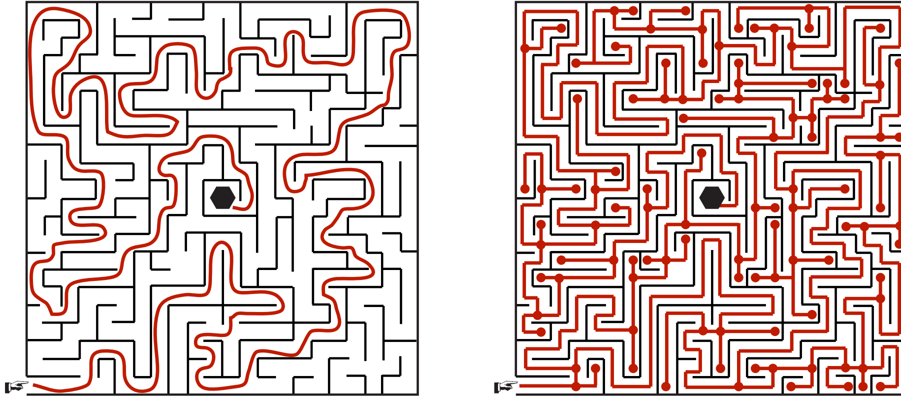
კუნძულ კრეტაზე ლაბირინთში დამწყვდეული იყო ადამიანის ტანისა და ხარის თავის მქონე ურჩხული მინოტავრი, რომლისთვის ათენელებს ხალხის მსხვერპლი უნდა შეეგზავნათ. ამ საშინელებისაგან თავის დახსნის მიზნით ბერძენი გმირი თესევსი ლაბირინთში შევიდა და მინოტავრი განგმირა. რადგან ლაბირინთში გზის გაკვლევა საკმაოდ რთული საქმეა, მან მეფის ქალიშვილის - არიადნას მიერ მიცემული ძაფი გამოიყენა, რითაც ადვილად გამოაგნო გარეთ (აქედან მომდინარეობს გამოთქმა „მსჯელობის ძაფი დაკარგა", „ძაფი გაექცა" და სხვა).



ნახ. 1.1: მსოფლიოში ყველაზე განთქმული ლაბირინთი

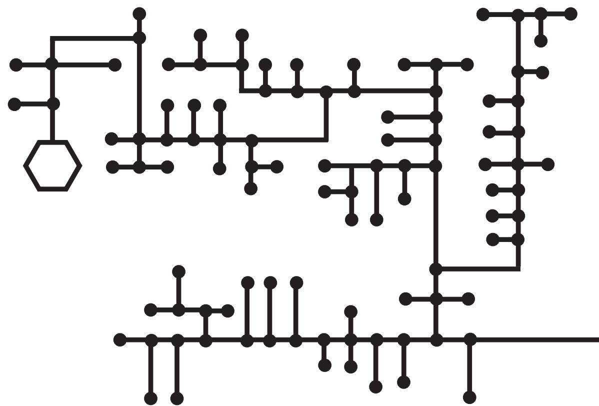
ეს მითი - ფილოსოფიური, ისტორიული, კულტურული და მრავალი სხვა მნიშვნელობის გარდა - ასევე დიდ როლს თამაშობს ინფორმატიკაშიც, რადგან უცნობ გარემოში მოძრაობის ამოცანას უკავშირდება, ხოლო ძაფის ან სხვა საშუალებებით განვლილი გზის მონიშვნა და უკან დაბრუნება ფართოდ გამოიყენება ალგორითმებთან დაკავშირებული პრობლემების გადასაჭრელად.

ყოველ ლაბირინთს შეგვიძლია შევუსაბამოთ რაღაცა გრაფი. ნახ. 1.2-ში ნაჩვენებია შედარებით რთული ლაბირინთი, რომელშიც შესაბამისი გრაფია ჩახაზული.



ნახ. 1.2: შედარებით რთული ლაბირინთი შესაბამისი გრაფით

იგივე გრაფი შეგვიძლია სხვანაირადაც დავხატოთ, რომ აღსაქმელად უფრო ადვილი იყოს (ნახ. 1.3). ცხადია, რომ თუ გვექნება ალგორითმი, რომლის საშუალებითაც გრაფის ყველა წვეროს შემოვლას შევძლებთ, ამით ლაბირინთში შესვლის ან გამოსვლის ალგორითმსაც ავაგებთ.



ნახ. 1.3: ლაბირინთის ექვივალენტური გრაფი

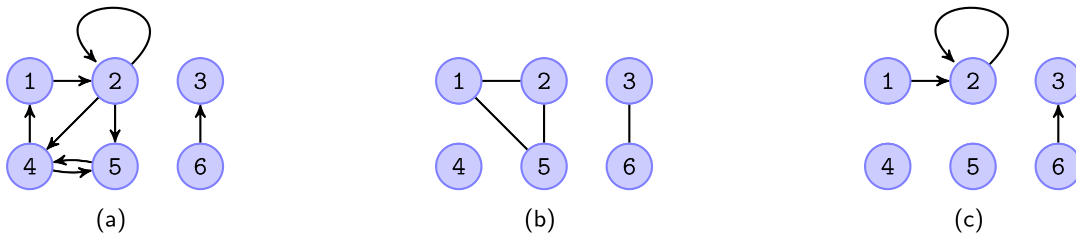
აღსანიშნავია, რომ ზედა ორ ნახაზში მოყვანილი გრაფი ერთმანეთის ექვივალენტურია (ერთის მეორეში გადაყვანა შეიძლება ისე, რომ გრაფის სტრუქტურა არ შეიცვალოს, აქ მხოლოდ დახატვის წესია სხვადასხვა), ამიტომ თუ ლაბირინთში შესვლისას გზის პირველ გასაყართან უნდა გავუხვიოთ მარცხნივ, მეორე გრაფში უნდა ვიაროთ პირდაპირ. მაგრამ ამას რაიმე პრინციპული მნიშვნელობა არ აქვს: ერთ გრაფზე მოძებნილი ამონახსნი ადვილად შეიძლება გადავიტანოთ მეორეზე.

ლაბირინთებში გზის გაკვლევის გარდა, გრაფში წვეროების შემოვლის ამოცანას მრავალი გამოყენება შეიძლება მოვუძებნოთ, მაგალითად, გრაფის წვეროების შიგთავსის ამობეჭდვაში, გრაფთა კოპირებასა ან სხვადასხვა ფორმატებში ჩაწერაში, წვეროთა ან წიბოთა რაოდენობის დათვლაში, გრაფის ბმული კომპონენტების პოვნაში, ორ წვეროს შორის გზის პოვნაში, გრაფში ციკლების აღმოჩენაში და ბევრ სხვა ამოცანაში.

თუ მოვახერხებთ იმას, რომ გრაფის შემოვლისას ყოველ წიბოზე გავიველით ***ზუსტად ორჯერ*** („იქით-აქეთ"), გვექნება იმის გარანტია, რომ არ გავიჭედებით და ყველა წვეროსაც გავივლით.

<u>სავარჯიშო 1.1:</u> ფორმალურად დაამტკიცეთ, რომ თუ გრაფის ყველა წიბოს შემოვუვლით ზუსტად ორჯერ, მის ყველა წვეროში ერთხელ მაინც შევალთ.

ახლა კი მოვიყვანოთ გრაფთა თეორიის ძირითადი განსაზღვრებები.
ორიენტირებული გრაფი (directed) G განისაზღვრება როგორც (V,E) წყვილი, სადაც V სასრული სიმრავლეა, ხოლო E წარმოადგენს V-ს ელემენტთა ბინარულ დამოკიდებულებას, ანუ $V \times V$ სიმრავლის ქვესიმრავლეა. ორიენტირებულ გრაფს ზოგჯერ **ორგრაფს** (digraph) უწოდებენ. V სიმრავლეს უწოდებენ გრაფის **წვეროთა სიმრავლეს** (vertex set), ხოლო E-ს - **წიბოთა სიმრავლეს** (edge set). მათ ელემენტებს შესაბამისად ეწოდებათ **წვერო** (vertex) და **წიბო** (edge). წიბოს, რომელიც წვეროს საკუთარ თავთან აერთებს, უწოდებენ **მარყუჟს (ციკლურ წიბოს)**.
არაორიენტირებულ გრაფში (undirected graph) G=(V,E) წიბოთა E სიმრავლე შედგება წვეროთა დაულაგებელი (unordered) წყვილებისაგან. წიბოს აღსანიშნავად გამოიყენება ჩანაწერი (u,v). არაორიენტირებულ გრაფში (u,v) და (v,u) ერთი და იგივე წიბოს აღნიშნავს, ხოლო მარყუჟი არ შეიძლება არსებობდეს, რადგან წიბო ორი განსხვავებული წვეროსაგან უნდა შედგებოდეს.



ნახ. 1.4:

სურ. 1.4ა-ზე მოცემულია ორიენტირებული გრაფი 6 წვეროთი და 8 წიბოთი V={1, 2, 3, 4, 5, 6} და E={(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)}. სურ. 1.4ბ-ზე - არაორიენტირებული გრაფი 6 წვეროთი და 4 წიბოთი V={1, 2, 3, 4, 5, 6} და E={(1,2), (1,5), (2,5), (3,6)}. (u,v) წიბოს შესახებ ორიენტირებულ გრაფში იტყვიან, რომ იგი გამოდის u წვეროდან და შედის v წვეროში. სურ. 1.4ა-ზე 2 წვეროდან გამოდის სამი წიბო - (2,2), (2,4), (2,5) და 2 წვეროში შედის ორი წიბო - (1,2), (2,2). არაორიენტირებულ გრაფში (u,v) წიბოს შესახებ იტყვიან, რომ იგი u და v წვეროების **ინციდენტურია** (incident).
თუ G გრაფში არსებობს (u,v) წიბო, იტყვიან, რომ v წვერო u წვეროს **მოსაზღვრეა** (is adjacent to u) არაორიენტირებულ გრაფებში მოსაზღვრეობა სიმეტრიული მიმართებაა, ხოლო ორიენტირებულ გრაფებისთვის ეს დებულება არ არის სამართლიანი. თუკი ორიენტირებულ გრაფში v წვერო u წვეროს მოსაზღვრეა, წერენ $u \rightarrow v$.
არაორიენტირებულ გრაფში წვეროს **ხარისხს** (degree) უწოდებენ ამ წვეროსადმი ინციდენტური წიბოების რაოდენობას. მაგალითად სურ. 1.4ბ-ზე 2 წვეროს ხარისხია 2. წვეროს, რომლის ხარისხიც არის 0, ეწოდება **იზოლირებული** (isolated) წვერო. წვეროს, რომლის ხარისხი არის 1 ეწოდება **დაკიდული** წვერო. ორიენტირებულ გრაფში განასხვავებენ **შემავალ** (in-degree) და **გამომავალ** (out-degree) ხარისხებს (შესაბამისად წვეროში შემავალი და გამომავალი წიბოების რაოდენობის მიხედვით) და მათ ჯამს უწოდებენ წვეროს ხარისხს. მაგალითად, სურ. 1.4ა-ზე 2 წვეროს ხარისხია 5 (შემავალი ხარისხი - 2, გამომავალი ხარისხი - 3). წვეროს, რომლის გამომავალი ხარისხი ნულია, ეწოდება **ჩასადენი** (sink), წვეროს, რომლის შემავალი ხარისხი ნულია, ეწოდება **წყარო** (source).
k **სიგრძის გზა (მარშრუტი)** (path of length k) u წვეროდან v წვეროში განიფაზღვრება როგორც წვეროთა $< v_0, v_1, v_2, \ldots, v_k >$ მიმდევრობა, სადაც $v_0 = u$, $v_k = v$ და $(v_{i-1}, v_i) \in E$ ნებისმიერი i=1,2,. . . ,k-სათვის. გზის სიგრძე განისაზღვრება მასში შემავალი წიბოების რაოდენობით. გზა შეიცავს (contains) $v_0, v_1, v_2, \ldots, v_k$ წვეროებს და $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ წიბოებს. ყოველთვის არსებობს ნული სიგძის გზა წვეროდან თავის თავში. v_0 წვეროს უწოდებენ გზის **დასაწყისს**, ხოლო v_k წვეროს - გზის **ბოლოს** და ამბობენ, რომ გზა მიდის v_0-დან v_k-საკენ. თუ მოცემული u და u' წვეროებისთვის არსებობს p გზა u-დან u'-ში, მაშინ ამბობენ, რომ u' მიღწევადია u-დან p გზით (u' is reachable from u via p). გზას ეწოდება **მარტივი** (simple), თუკი ყველა წვერო მასში განსხვავებულია. სურ. 1.4ა-ზე 3 სიგრძის მქონე გზა ¡1, 2, 5, 4¿ მარტივია, ხოლო იმავე სიგრძის გზა ¡2, 5, 4, 5¿ - არაა მარტივი.
$p =< v_0, v_1, v_2, \ldots, v_k >$ გზის **ქვეგზა** (subpath) ეწოდება ამ გზიდან მიყოლებით აღებულ $< v_i, v_{i+1}, \ldots, v_j >$ წვეროთა მიმდევრობას, რომლისთვისაც $0 \leq i \leq j \leq k$. ორიენტირებულ გრაფში **ციკლი** (cycle) ეწოდება გზას, რომელშიც საწყისი წვერო ემთხვევა ბოლო წვეროს და რომელიც ერთ წიბოს მაინც შეიცავს. ციკლს ეწოდება **მარტივი**, თუკი მასში არ მეორდება არც ერთი წვერო პირველისა და ბოლოს გარდა. **მარყუჟი** წარმოადგენს ციკლს სიგრძით 1. აიგივებენ ციკლებს, რომლებიც განსხვავდებიან მხოლოდ წანაცვლებით ციკლის გასწვრივ. მაგ.: სურ. 1.4ა-ზე გზები ¡1, 2, 4, 1¿, ¡2, 4, 1, 2¿ და ¡4, 1, 2, 4¿ წარმოადგენენ ერთსა და იმავე ციკლს. ამავე ნახაზზე ციკლი ¡2, 2¿ შექმნილია ერთი წიბოთი. ორიენტირებულ გრაფს უწოდებენ **მარტივს**, თუკი იგი არ შეიცავს მარყუჟებს.

არაორიენტირებულ გრაფში $< v_0, v_1, v_2, \ldots, v_k >$ გზას ეწოდება **მარტივი ციკლი**, თუ $k \geq 3$, $v_0 = v_k$ და ყველა $v_1, v_2, \ldots, v_k$ წვერო განსხვავებულია. სურ. 1.4ბ-ზე მარტივი ციკლია ¡1, 2, 5, 1¿. გრაფს, რომელშიც არაა ციკლები, **აციკლური** (acyclic) ეწოდება.

არაორიენტირებულ გრაფს ეწოდება **ბმული** (connected), თუკი წვეროთა ნებისმიერი წყვილისათვის არსებობს გზა ერთიდან მეორეში. არაორიენტირებულ გრაფში გზის არსებობა ერთი წვეროდან მეორეში წარმოადგენს ეკვივალენტურ მიმართებას წვეროთა სიმრავლეზე. ეკვივალენტობის კლასებს ეწოდებათ გრაფის **ბმული კომპონენტები** (connected components). მაგ.: სურ. 1.4ბ-ზე სამი ბმული კომპონენტია: {1, 2, 5}, {3, 6} და {4}. არაორიენტირებული გრაფი ბმულია მაშინ და მხოლოდ მაშინ, როცა ის შედგება ერთადერთი ბმული კომპონენტისაგან.

ორიენტირებულ გრაფს ეწოდება **ძლიერად ბმული** (strongly connected), თუკი მისი ნებისმიერი წვეროდან მიღწევადია (ორიენტირებული გზებით) ნებისმიერი სხვა წვერო. ნებისმიერი ორიენტირებული გრაფი შეიძლება დაიყოს **ძლიერად ბმულ კომპონენტებად** (strongly connected components). სურ. 1.4ა-ზე არის სამი ასეთი კომპონენტი {1, 2, 4, 5}, {3} და {6}. შევნიშნოთ, რომ 3 და 6 წვეროები ერთად არ ჰქმნიან ძლიერად ბმულ კომპონენტს, რადგან არ არსებობს გზა 6-დან 3-ში.

G=(V,E) და G'=(V',E') გრაფებს ეწოდებათ **იზომორფულები** (isomorphic), თუკი არსებობს ურთიერთცალსახა შესაბამისობა $f : V \to V'$ მათი წვეროების სიმრავლეებს შორის, რომლის დროსაც $(u, v) \in E$ მაშინ და მხოლოდ მაშინ, როცა $(f(u), f(v)) \in E'$. შეიძლება ითქვას, რომ იზომორფული გრაფები ეს ერთი და იგივე გრაფია, სადაც წვეროები სხვადასხვაგვარადაა დასახელებული.

G'=(V',E') გრაფს ეწოდება G=(V,E) გრაფის **ქვეგრაფი** (subgraph), თუ $V' \subseteq V$ და $E' \subseteq E$. თუ G=(V,E) გრაფში ავირჩევთ V' წვეროთა ნებისმიერ სიმრავლეს, მაშინ შეგვიძლია განვიხილოთ G-ს ქვეგრაფი, რომელიც შედგება ამ წვეროებისა და მათი შემაერთებელი წიბოებისაგან. ამ ქვეგრაფს უწოდებენ G გრაფის **შეზღუდვას** V' წვეროთა სიმრავლეზე. სურ. 1.4ა-ზე მოცემული გრაფის შეზღუდვა {1, 2, 3, 6} წვეროთა სიმრავლეზე ნაჩვენებია სურ. 1.4ც-ზე და შეიცავს სამ წიბოს (1,2), (2,2), (6,3).

ნებისმიერი არაორიენტირებული გრაფისათვის შეიძლება განვიხილოთ მისი **ორიენტირებული ვარიანტი** (directed version), თუკი ყოველ (u,v) არაორიენტირებულ წიბოს შევცვლით ორიენტირებული წიბოების (u,v) და (v,u) წყვილით, რომლებსაც ექნებათ ურთიერთსაწინააღმდეგო მიმართულებები. მეორე მხრივ, ნებისმიერი ორიენტირებული გრაფისათვის შეიძლება განვიხილოთ მისი **არაორიენტირებული ვარიანტი** (undirected version), თუკი ამოვშლით მარყუჟებს და (u,v) და (v,u) წიბოებს შევცვლით არაორიენტირებული (u,v) წიბოთი. ორიენტირებულ გრაფში წვეროს **მეზობელი** (neighbor) ეწოდება ნებისმიერ წვეროს, რომელიც შეერთებულია მასთან ნებისმიერი მიმართულების წიბოთი, ე.ი. v წვერო არის u-ს მეზობელი თუ v არის u-ს მოსაზღვრე ან u არის v-ს მოსაზღვრე. არაორიენტირებულ გრაფში კი ცნებები "მეზობელი" და "მოსაზღვრე" სინონიმებია.

სრული (complete) გრაფი ეწოდება არაორიენტირებულ გრაფს, რომელიც შეიცავს ყველა შესაძლებელ წიბოს წვეროთა მოცემული სიმრავლისათვის, ე.ი. ნებისმიერი წვერო შეერთებულია ყველა დანარჩენთან. არაორიენტირებულ G = (V,E) გრაფს უწოდებენ **ორწილას** (bipartite), თუ V წვეროთა სიმრავლე შეიძლება გავყოთ ისეთ ორ V_1 და V_2 ნაწილად, რომ ნებისმიერი წიბოს ბოლოები სხვადასხვა ნაწილში აღმოჩნდეს. G=(V,E) გრაფის **დამატება** ეწოდება გრაფს, რომელსაც წვეროთა იგივე სიმრავლე აქვს, ხოლო ორი წვერო მოსაზღვრეა მაშინ და მხოლოდ მაშინ, როცა ისინი არ არიან მოსაზღვრე წვეროები G გრაფში. აციკლურ არაორიენტირებულ გრაფს უწოდებენ **ტყეს** (forest), ხოლო ბმულ აციკლურ არაორიენტირებულ გრაფს უწოდებენ (**თავისუფალ**) **ხეს** (free tree). **ორიენტირებული აციკლური გრაფის** (directed acyclic graph) აღსანიშნავად ზოგჯერ იყენებენ მის აბრევიატურას - DAG.

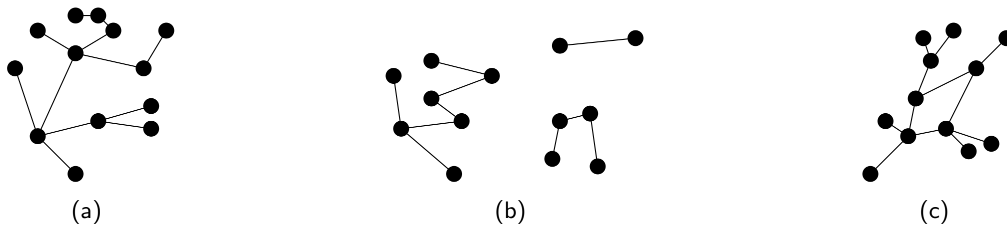
თუ გრაფის წვეროების ხარისხების საშუალო მნიშვნელობა $2|E|/|V|$ ახლოსაა $|V|$-სთან, ან, სხვა სიტყვებით რომ ვთქვათ, $|E|$ იმავე რიგისაა, რაც $|V|^2$, გრაფს ეწოდება **მკვრივი** (dence) გრაფი. **ხალვათი** (sparse) ეწოდება გრაფს, რომელშიც $|E|$ მკვეთრად მცირეა $|V|^2$-თან შედარებით. სხვა განმარტებით, **ხალვათი** ეწოდება გრაფს, რომლის დამატებაც მკვრივია.

1.3 ხეები

როგორც ზემოთ აღვნიშნეთ, ბმულ აციკლურ არაორიენტირებულ გრაფს უწოდებენ **ხეს** ან **თავისუფალ ხეს** (free tree), ხოლო აციკლურ არაორიენტირებულ გრაფს უწოდებენ ტყეს (forest). ტყე შედგება ხეებისაგან, რომლებიც მის ბმულ კომპონენტებს წარმოადგენენ. ხეებისათვის ვარგისი ბევრი ალგორითმი გამოსადეგია ტყეებისთვისაც. სურ. 1.5ა-ზე გამოსახულია ხე, სურ. 1.5ბ-ზე - ტყე (ის არ წარმოადგენს ხეს იმის გამო, რომ ბმული არაა), ხოლო სურ. 1.5ც მოცემული გრაფი არც ხეა და არც ტყე, რადგან იგი ციკლს შეიცავს.

თეორემა 1.1. (ხეთა თვისებები). ვთქვათ, G=(V,E) არაორიენტირებული გრაფია. მაშინ ტოლფასია შემდეგი თვისებები:

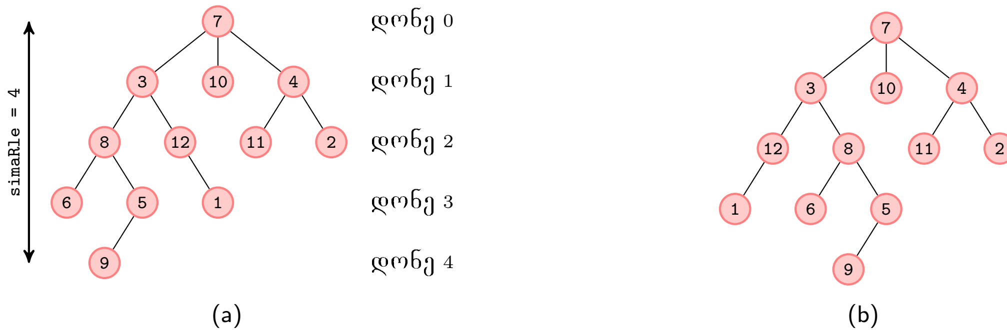
1. G ხეა



ნახ. 1.5:

2. G-ს ნებისმიერი ორი წვეროსათვის არსებობს მათი შემაერთებელი ერთადერთი მარტივი გზა
3. G გრაფი ბმულია, მაგრამ კარგავს ბმულობას, თუ ამოვიღებთ ნებისმიერ წიბოს
4. G გრაფი ბმულია და $|E| = |V| - 1$
5. G გრაფი აციკლურია და $|E| = |V| - 1$
6. G გრაფი აციკლურია, მაგრამ ნებისმიერი წიბოს დამატებით მასში ჩნდება ციკლი

ხე ფესვით (rooted tree) მიიღება მაშინ, როცა ბმულ აციკლურ არაორიენტირებულ გრაფში გამოყოფილია ერთ-ერთი წვერო, რომელსაც **ფესვს** (root) უწოდებენ. ხშირად, ფესვის მქონე ხის წვეროებს **კვანძებს** (nodes) უწოდებენ. სურ. 1.6-ზე ნაჩვენებია ფესვის მქონე ხე 12 წვეროთი და ფესვით 7.



ნახ. 1.6:

ვთქვათ x არის r ფესვის მქონე T ხის კვანძი. ნებისმიერ y კვანძს, რომელიც მდებარეობს (ერთადერთ) გზაზე r-დან x-ში, უწოდებენ x კვანძის **წინაპარს** (ancestor). თუ y არის x-ის წინაპარი, მაშინ x-ს უწოდებენ y-ის **შთამომავალს** (descendant). ყოველი კვანძი შეიძლება ჩაითვალოს საკუთარ წინაპრად ან შთამომავლად. თუ y არის x-ის წინაპარი და $x \neq y$, მაშინ y-ს უწოდებენ **საკუთარ წინაპარს** (proper ancestor), ხოლო x-ს უწოდებენ **საკუთარ შთამომავალს** (proper descendant).

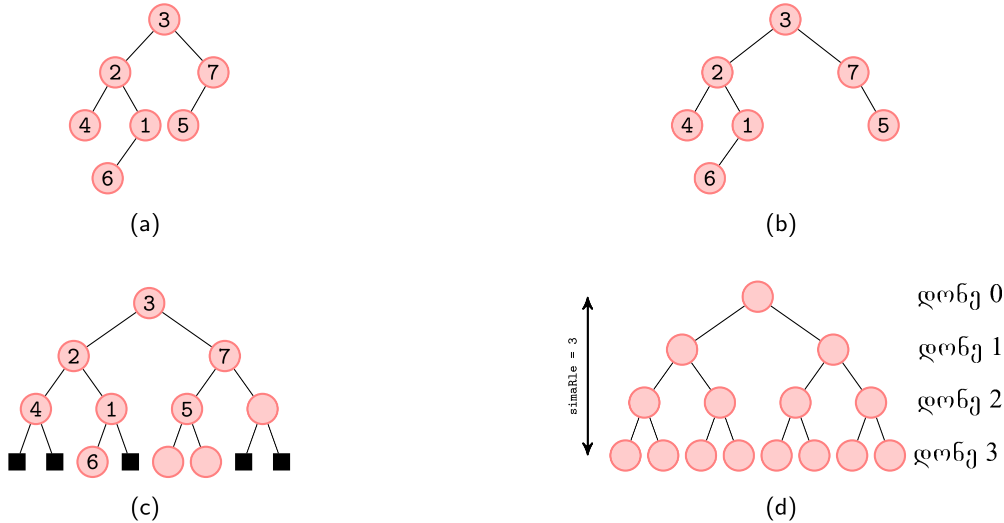
ყოველი x კვანძისათვის შეიძლება განვიხილოთ ხე, რომელიც x-ის ყველა შთამომავლისაგან შედგება და x ითვლება ამ ხის ფესვად. მას უწოდებენ **ქვეხეს** x **ფესვით**. მაგალითად სურ. 1.6ა-ზე ქვეხე ფესვით 8 შეიცავს წვეროებს 8, 6, 5 და 9.

თუ (y,x) უკანასკნელი წიბოა T ხეში r ფესვიდან x-საკენ მიმავალ გზაზე, მაშინ y-ს უწოდებენ x-ის **მშობელს** (parent), ხოლო x-ს უწოდებენ y-ის **შვილს** (child). ფესვი ერთადერთი კვანძია, რომელსაც მშობელი არ ჰყავს. კვანძებს, რომელთაც საერთო მშობელი ჰყავს, უწოდებენ **დედმამიშვილებს** (siblings, რუსულ ლიტერატურაში - ძმები). ფესვის მქონე ხის კვანძს, რომელსაც არა ჰყავს შვილები, უწოდებენ **ფოთოლს** (leaf, external node). წვეროებს, რომელთაც ჰყავთ შვილები, უწოდებენ **შინაგანს** (internal). ფესვის მქონე ხის კვანძის შვილების რაოდენობას უწოდებენ მის **ხარისხს** (degree). გზის სიგრძეს ფესვიდან ნებისმიერ x კვანძამდე უწოდებენ x წვეროს **დონეს** (depth). ხის კვანძის **სიმაღლე** (height) არის წიბოების რაოდენობა ყველაზე გრძელ გზაზე ამ კვანძიდან მის შთამომავალ ფოთლამდე. ხის სიგრძედ ითვლება მისი ფესვის სიგრძე.

დალაგებული ხე (ordered tree) ეწოდება ფესვის მქონე ხეს, რომელსაც დამატებითი სტრუქტურა აქვს: ყოველი კვანძისათვის მისი შვილების სიმრავლე დალაგებულია (ცნობილია თუ რომელია წვეროს პირველი შვილი, მეორე შვილი და ა.შ.). სურ. 1.6-ზე გამოსახული ორ ხეს ერთი და იგივე ფესვი აქვს და ისინი განსხვავდებიან მხოლოდ შვილების დალაგებით.

ორობითი ხე (binary tree) ყველაზე მარტივი სახით განისაზღვრება რეკურსიულად, როგორც კვანძთა სასრული სიმრავლე, რომელიც: ან ცარიელია (არ შეიცავს კვანძებს), ან გაყოფილია სამ არაგადამკვეთ ნაწილად: წვერო

რომელსაც ეწოდება **ფესვი** (root), ორობითი ხე, რომელსაც ეწოდება ფესვის **მარცხენა ქვეხე** (left subtree) და ორობითი ხე, რომელსაც ეწოდება ფესვის **მარჯვენა ქვეხე** (right subtree) ორობით ხეს, რომელიც არ შეიცავს კვანძებს, ეწოდება **ცარიელი** (empty). ზოგჯერ მას აღნიშნავენ NIL-ით. თუ მარცხენა ქვეხე არაცარიელია, მაშინ მის ფესვს უწოდებენ მთლიანი ხის ფესვის **მარცხენა შვილს** (left child), შესაბამისად განისაზღვრება **მარჯვენა შვილიც** (right child). ორობითი ხის მაგალითი მოცემულია სურ. 1.7-ზე.
არასწორი იქნებოდა განგვესაზღვრა ორობითი ხე, როგორც დალაგებული ხე, რომელშიც თითოეული კვანძის ხარისხი არ აღემატება 2-ს. ამის მიზეზია ის, რომ ორობით ხეში მნიშვნელობა აქვს როგორია კვანძის ერთადერთი შვილი - მარცხენა თუ მარჯვენა, ხოლო დალაგებული ხისათვის ასეთი განსხვავება არ არსებობს. სურ. 1.7ა-ზე და სურ. 1.7ბ-ზე ნაჩვენები ორობითი ხეები განსხვავდებიან, რადგან 1.7ა-ზე 5 არის 7-ის მარცხენა შვილი, ხოლო 1.7ბ-ზე - მარჯვენა. როგორც დალაგებული ხეები ისინი ერთნაირები არიან.



ნახ. 1.7:

ხშირად ორობით ხეზე ცარიელ ადგილებს ავსებენ ფიქტიური ფოთლებით, მიიღება **სრულად ორობითი ხე** (full binary tree). ამის შემდეგ ყველა კვანძი ან ფოთოლია, ან აქვს ხარისხი 2, 1-ს ტოლი ხარისხის მქონე კვანძები ასეთ ხეში არ არის. ეს გარდაქმნა ნაჩვენებია სურ. 1.7ც-ზე.
სრული k -ობითი ხე (complete k-ary tree) ეწოდება k-ობით ხეს, რომელშიც ყველა ფოთოლს აქვს ერთნაირი დონე და ყველა შინაგან კვანძს აქვს ხარისხი k. ასეთ შემთხვევაში ხის სტრუქტურა მთლიანად განისაზღვრება მისი სიმაღლით. სურ. 1.7დ-ზე გამოსახულია სრული ორობითი ხე სიმაღლით 3. ფესვი ყოველთვის არის 0 დონის ერთადერთი კვანძი, მისი k შვილი არის 1 დონის მქონე კვანძები, რომელთა k^2 შვილი წარმოადგენენ 2 დონის კვანძებს და ა.შ. h დონეზე გვექნება k^h ფოთოლი. h სიმაღლის სრული k-ობითი ხის შინაგანი კვანძების რაოდენობა ტოლია:

$$1 + k + k^2 + \cdots + k^{h-1} = \frac{k^h - 1}{k - 1}$$

კერძოდ, სრული ორობითი ხის შინაგანი კვანძების რაოდენობაა 2^h-1.

1.4 გრაფის წარმოდგენა

გრაფის წარმოდგენისთვის გამოიყენება სამი ძირითადი სტრუქტურა:

1. **მოსაზღვრე წვეროების სია** (adjacency list representation)
2. **მოსაზღვრეობის მატრიცა** (adjacency matrix representation)
3. **წიბოების სია** (edge list representation)

G=(V,E) გრაფის წარმოდგენა მოსაზღვრე წვეროების სიებით იყენებს V ცალი წვეროსგან შემდგარ ბმულ სიას. მოსაზღვრე წვეროთა ყველა სიის სიგრძეთა ჯამი ორიენტირებული გრაფისათვის წიბოთა რაოდენობის ტოლია, ხოლო არაორიენტირებული გრაფისათვის - წიბოთა გაორმაგებული რაოდენობის ტოლი, რადგან (u,v) წიბო წარმოშობს ელემენტს როგორც u, ასევე v წვეროს მოსაზღვრე წვეროთა სიაში. ორივე შემთხვევაში მეხსიერების

საჭირო მოცულობაა O(V+E). ამ წარმოდგენით მოსახერხებელია წონადი გრაფების (weighted graphs) შენახვაც, სადაც ყოველ წიბოს შეესაბამება ნამდვილი რიცხვი - ამ წიბოს **წონა** (weight), ანუ მოცემულია **წონის ფუნქცია** (weight function) $w : E \to R$. ამ შემთხვევაში მოსახერხებელია $(u, v) \in E$ წიბოს w(u,v) წონის შენახვა u წვეროსთან ერთად წვეროს მოსაზღვრე წვეროთა სიაში. თუმცა ამ წარმოდგენას აქვს მნიშვნელოვანი ნაკლი: u-დან v-ში წიბოს არსებობის დასადგენად, საჭიროა გადავამოწმოთ მთლიანი სია, მასში v-ს მოსაძებნად. ძებნას შესაძლოა თავი ავარიდოთ, თუ გამოვიყენებთ მოსაზღვრეობის მატრიცას, თუმცა ამ შემთხვევაში მეტი მანქანური მეხსიერებაა საჭირო.
მოსაზღვრეობის მატრიცის გამოყენებისას უნდა გადავნომროთ G=(V,E) გრაფის წვეროები 1,2,. . . ,V რიცხვებით და განვიხილოთ $|V| \times |V|$ ზომის $A = (a_{ij})$ მატრიცა, სადაც $a_{ij} = 1$, თუ $(i, j) \in E$ და $a_{ij} = 0$, წინააღმდეგ შემთხვევაში. მეხსიერების საჭირო მოცულობაა O(V^2). არაორიენტირებული გრაფისათვის მოსაზღვრეობის მატრიცა სიმეტრიულია მთავარი დიაგონალის მიმართ და ამიტომ საკმარისია შევინახოთ მხოლოდ მთავარი დიაგონალი და მის ზემოთ მყოფი ელემენტები, რაც თითქმის ორჯერ ამცირებს გამოყენებულ მეხსიერებას. წონადი გრაფების შენახვა პრობლემა არც ამ მეთოდისთვისაა - (u,v) წიბოს w(u,v) წონა მატრიცაში თავსდება u სტრიქონისა და v სვეტის გადაკვეთაზე, ხოლო წიბოს არარსებობის შემთხვევაში მატრიცის შესაბამისი ელემენტი მიიღებს ცარიელ მნიშვნელობას NIL (ზოგ ამოცანაში შეიძლება გამოვიყენოთ 0 ან ∞).
სურ. 1.8-ზე მოცემულია გრაფთა წარმოდგენის ორივე მეთოდი როგორც არაორიენტირებული, ისევე ორიენტირებული გრაფებისათვის.
თითოეული წარმოდგენის გამოყენების შესახებ გადაწყვეტილების მიღება დამოკიდებულია: გრაფის განზომილებაზე (მოსაზღვრეობის მატრიცით წარმოდგენა მოსახერხებელია პატარა ან მკვრივი გრაფებისათვის), ალგორითმზე (მოსაზღვრეობის მატრიცით წარმოდგენა უმჯობესია ალგორითმებისთვის, რომლებიც პასუხობენ კითხვაზე, მაგ.: არის თუ არა (u,v) წიბო G გრაფში?) იმაზე, ხალვათია თუ მკვრივი გრაფი (თუ გრაფი მკვრივია, მატრიცით წარმოდგენა უფრო მოსახერხებელია, რადგან ყველა შემთხვევაში O(V^2) მეხსიერების გამოყენება მოგვიწევს).
თუკი მანქანური მეხსიერება საკმარისია, სჯობს გრაფი აღვწეროთ მოსაზღვრეობის მატრიცის საშუალებით, რადგან უმეტეს შემთხვევაში მასთან მუშაობა უფრო მოსახერხებელია. ამას გარდა, თუკი გრაფი წონადი არ არის, მოსაზღვრეობის მატრიცის ელემენტები შესაძლოა განიხილოთ როგორც ბიტები, რომლებიც შეგვიძლია გავაერთიანოთ მანქანურ სიტყვებში - ეს იძლევა მეხსიერების მნიშვნელოვან ეკონომიას.
გრაფისთვის ეფექტური ალგორითმის შერჩევის დროს, ერთ-ერთი მთავარი ფაქტორია ინფორმაცია იმის შესახებ ხალვათია თუ მკვრივი გრაფი. მაგ.: თუ რაიმე ამოცანის გადასაწყვეტად შეგვიძლია შევიმუშაოთ 2 ალგორითმი: პირველს სჭირდება —V—^{2}, ხოლო მეორეს - $|E|\log|E|$ ბიჯი. ეს ფორმულები გვიჩვენებს, რომ პირველი ალგორითმი უფრო გამოდგება მკვრივი გრაფებისთვის, ხოლო მეორე - ხალვათისთვის. მაგ.: განვიხილოთ მკვრივი გრაფი —E—=10^6 და —V—=10^3, —V—^{2} 20-ჯერ ჩქარია $|E|\log|E|$-ზე. მეორე მხრივ, ხალვათი გრაფისთვის, —E—=10^6 და —V—=10^6 ალგორითმი $|E|\log|E|$ სირთულით მილიონის რიგით აჯობებს —V—^{2} სირთულის ალგორითმს.

1.5 სავარჯიშოები

1. ოთხი ადამიანი ღამით მოძრაობს გზაზე ერთი მიმართულებით. მათ უნდა გადაიარონ ხიდზე და აქვთ მხოლოდ 1 ფარანი. ხიდზე ერთდროულად შეუძლია გაიაროს არაუმეტეს 2 ადამიანმა (ან ერთმა, ან ორმა) და ერთ-ერთს აუცილებლად უნდა ეჭიროს ფარანი. არ შეიძლება ფარანის ერთი ნაპირიდან მეორეზე გადაგდება, შეიძლება მისი მხოლოდ ხიდით გადატანა. თითოეული ადამიანი ხიდის გადასვლას უნდება: პირველი-1წთ, მეორე-2წთ, მესამე-5წთ, მეოთხე-10წთ. თუ ხიდზე გადადის 2 ადამიანი, ისინი მოძრაობენ უფრო ნელი სიჩქარით. რა თანმიმდევრობით უნდა გადაიარონ ხიდი, რომ ამისთვის დასჭირდეთ ზუსტად 17 წუთი.

2. მოიყვანეთ შემდეგი გრაფების მაგალითები ან აჩვენეთ, რომ ასეთი გრაფები არ არსებობს:

   (ა) გრაფი, რომელსაც აქვს ჰამილტონის ციკლი, მაგრამ არ აქვს ეილერის ციკლი

   (ბ) გრაფი, რომელსაც აქვს ეილერის ციკლი, მაგრამ არ აქვს ჰამილტონის ციკლი

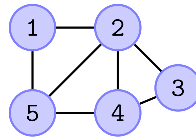
   (ც) გრაფი, რომელსაც აქვს როგორც ეილერის, ისე ჰამილტონის ციკლი

   (დ) გრაფი, რომელსაც აქვს ყველა წვეროს შემცველი ციკლი, მაგრამ არ აქვს არც ჰამილტონის, არც ეილერის ციკლი

3. დაამტკიცეთ, რომ V წვეროს შემცველი არაორიენტირებული გრაფი შეიცავს არაუმეტეს $|V|(|V|-1)/2$ წიბოს.

4. დაამტკიცეთ, რომ ნებისმიერი G=(V,E) არაორიენტირებული ბმული გრაფისთვის სრულდება $|E| \geq |V| - 1$.

5. დაამტკიცეთ, რომ ნებისმიერი G=(V,E) აციკლური ბმული არაორიენტირებული გრაფი შეიცავს $|V| - 1$ წიბოს.

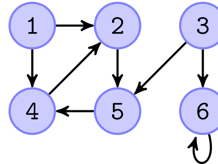


1	2, 5
2	1, 3, 4, 5
3	2, 4
4	2, 3, 5
5	1, 2, 4

(a) მოსაზღვრე წვეროების სია

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(b) მოსაზღვრეობის მატრიცა



1	2, 4
2	5
3	5, 6
4	2
5	4
6	6

(c) მოსაზღვრე წვეროების სია

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

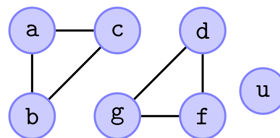
(d) მოსაზღვრეობის მატრიცა

ნახ. 1.8:

6. (ა) გრაფის მოსაზღვრეობის მატრიცის რა თვისებები მეტყველებს იმაზე, რომ:
   i. გრაფი სრულია
   ii. გრაფი შეიცავს მარყუჟს
   iii. გრაფი შეიცავს იზოლირებულ წვეროს

   (ბ) გაეცით პასუხები (ა)-ში დასმულ შეკითხვებზე გრაფის მოსაზღვრე წვეროთა სიით წარმოდგენის შემთხვევისთვის.

7. იპოვეთ ეკვივალენტობის კლასები შემდეგი გრაფისათვის:

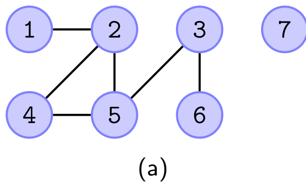


8. შეამოწმეთ, რომ მიმართება "მიღწევადია -დან" არაორიენტირებულ გრაფში არის ეკვივალენტობის მიმართება გრაფის წვეროთა სიმრავლეზე.

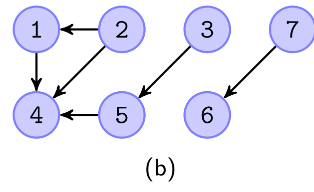
9. ეკვივალენტობის მიმართების 3 თვისებიდან რომელი ირღვევა "მიღწევადია -დან" მიმართებისთვის ორიენტირებულ გრაფში?

10. რამდენი ბმული კომპონენტი აქვს ხეს?

11. შემდეგი გრაფებისთვის ააგეთ მოსაზღვრეობის მატრიცა და მოსაზღვრე წვეროთა სია.



(a)



(b)

თავი 2

გრაფის შემოვლის ალგორითმები

ბევრი ალგორითმი, რომელიც მუშაობს გრაფებზე, საჭიროებს გრაფის წვეროების და წიბოების გარკვეულ დამუშავებას. გრაფების შემოვლისთვის არსებობს ორი ძირითადი ალგორითმი - **სიგანეში ძებნა** (breadth-first search, BFS) და **სიღრმეში ძებნა** (depth-first search, DFS). ეს ალგორითმები გამოიყენება როგორც არაორიენტირებული, ისე ორიენტირებული გრაფებისთვის. გარდა მათი ძირითადი დანიშნულებისა (წვეროებისა და წიბოების შემოვლა), მათი გამოყენება სასარგებლოა გრაფების რიგი მნიშვნელოვანი თვისების დასადგენად (მაგ, გრაფის ბმულობის, აციკლურობის და ა.შ.)

2.1 სიგანეში ძებნის ალგორითმი

სიგანეში ძებნის ალგორითმი გრაფის შემოვლის ერთ-ერთი მარტივი ალგორითმია, რომელიც გრაფებთან მომუშავე ბევრი ალგორითმის საფუძველს წარმოადგენს. მაგალითად, მინიმალური დამფარავი ხის აგების პრიმის ალგორითმი, ერთი წვეროდან უმოკლესი მანძილის პოვნის დეიქსტრას ალგორითმი იყენებენ სიგანეში ძებნის ალგორითმის მსგავს იდეებს.
ვთქვათ მოცემულია G=(V,E) გრაფი და მისი s **საწყისი წვერო** (source vertex). სიგანეში ძებნის (breadth-first search) ალგორითმი ადგენს უმოკლეს მანძილს s-დან ყველა მიღწევად წვერომდე (მანძილად აქ ითვლება უმოკლეს გზაზე წიბოთა რაოდენობა, წიბოს წონა ერთის ტოლად ითვლება). ალგორითმის მუშაობის პროცესში მიიღება ე.წ. ძებნის ხე, რომელიც მოიცავს ფესვიდან მიღწევად ყველა წვეროს.
მეტი თვალსაჩინოებისათვის ჩავთვალოთ, რომ ალგორითმის მუშაობის პროცესში გრაფის წვეროები შესაძლოა იყოს თეთრი, რუხი ან შავი ფერის. თავდაპირველად ყველა წვერო თეთრი ფერისაა, ხოლო ალგორითმის მუშაობის დროს თეთრი წვერო შეიძლება გადაიქცეს რუხად, ხოლო რუხი - შავად (მაგრამ არა პირიქით). რუხად იღებება ის წვერო, რომელიც ალგორითმმა აღმოაჩინა, მაგრამ ჯერ არაა შემოწმებული მისგან გამომავალი სხვა წიბოები, ხოლო შავად იღებება ის წვერო, რომლისგანაც გამომავალი ყველა წიბო უკვე შემოწმებულია.
თავიდან ძებნის ხე შედგება მხოლოდ ფესვისაგან. როცა ალგორითმი პირველად აღმოაჩენს ახალ (თეთრი ფერის) v წვეროს, რომელიც უკვე ნაპოვნი u წვეროს მოსაზღვრეა, v წვერო ხდება u წვეროს **შვილი** (child) და იღებება რუხად, ხოლო (u,v) წიბო ემატება ძებნის ხეს. ასეთ წიბოს უწოდებენ **ხის წიბოს** (tree edge). u წვერო ხდება v წვეროს **მშობელი** (parent) და თუკი მისი ყველა შვილი ნაპოვნია, იღებება შავად. თუ წიბოს მივყავართ უკვე აღმოჩენილ წვეროსთან, რომელიც არ წარმოადგენს მის უშუალო წინაპარს, მას უწოდებენ **ჯვარედინ წიბოს** (cross edge). ყოველი წვეროს აღმოჩენა ხდება მხოლოდ ერთხელ, ამიტომ წვეროს არ შეიძლება ერთზე მეტი მშობელი ჰყავდეს. "წინაპრისა" და "შთამომავლის" ცნებებიც ამ შემთხვევაში ჩვეულებრივად განისაზღვრება.
პროცედურა BFS (breadth-first-search - განივად ძებნა) იყენებს გრაფის წარმოდგენას მოსაზღვრე წვეროთა სიით. ყოველი u წვეროსათვის დამატებით ინახება მისი ფერი color[u] და მისი მშობელი π[u]. თუკი მშობელი ჯერ ნაპოვნი არ არის ან u=s, მაშინ π[u]=NIL. მანძილი s-დან u-მდე იწერება d[u] ველში. რუხი წვეროების შესანახად გამოიყენება რიგი Q (განმარტება თავის ბოლოს).
2-5 სტრიქონებში ყველა წვეროსათვის ფერი ხდება თეთრი, მნიშვნელობბები - უსასრულობა, ხოლო მშობლები - NIL. მე-6 სტრიქონში ხდება ფესვის (s წვეროს) დამუშავება. მე-7 სტრიქონში Q ცარიელ რიგს ემატება მისი პირველი წევრი - s წვერო. პროგრამის ძირითადი ციკლი (8-16 სტრიქონები) სრულდება Q რიგის დაცარიელებამდე, ე.ი. სანამ არსებობენ რუხი ფერის წვეროები, ანუ წვეროები, რომლებიც აღმოჩენილია, მაგრამ რომელთა მოსაზღვრეობის სიები ჯერ განხილული არ არის. პირველი იტერაციის წინ, ერთადერთი რუხი ფერის წვერო და ერთადერთი წვერო არის საწყისი s წვერო. მე-9 სტრიქონით განისაზღვრება რუხი წვერო u რიგის თავში, რომელიც შემდეგ ამოიშლება Q რიგიდან. 10-15 სტრიქონებში for ციკლი განიხილავს u-ს ყველა მოსაზღვრე v წვეროს მოსაზღვრე წვეროთა სიაში. თუკი მათ შორის აღმოჩნდება თეთრი ფერის წვერო, ის იღებება რუხად, მის

Algorithm 1: Breadth First Search (BFS)

Input: გრაფი $G = (V, E)$ და საწყისი წვერო s
Output: გრაფის განივად ძებნისას აღმოჩენილი წვეროების მიმდევრობა

```
1  BFS(G, s) :
2      for ∀u ∈ V\{s} :
3          color[u] = TeTri;
4          d[u] = ∞;
5          π[u] = NIL;
6      color[s] = ruxi; d[s] = 0; π[s] = NIL; Q = ∅;
7      ENQUEUE(Q, s);
8      while Q ≠ ∅ :
9          u = DEQUEUE(Q);
10         for ∀v ∈ Adj[u] :
11             if color[v] == TeTri :
12                 color[v] = ruxi;
13                 d[v] = d[u] + 1;
14                 π[v] = u;
15                 ENQUEUE(Q, v);
16         color[u] = Savi;
17     return d, π
```

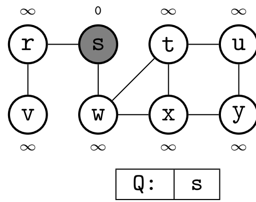
მშობლად ცხადდება u და მანძილად ფესვამდე - d[u]+1, ხოლო თავად ეს წვერო თავსდება Q რიგის ბოლოში. ამის შემდეგ u წვერო ილებება შავად (16 სტრიქონი).
სურ. 2.1-ზე მოცემულია BFS პროცედურის მუშაობა არაორიენტირებული გრაფისათვის და აგებულია სიგანეში ძებნის ხე. მუქად აღნიშნულია **ხის** წიბოები T (tree edges). წიბოები, რომლებიც არ შედიან სიგანეში ძებნის ხეში, არის **ჯვარედინი** წიბოები - C (cross edges).
სიგანეში ძებნის შედეგი შეიძლება დამოკიდებული იყოს u-ს მოსაზღვრე წვეროების თანმიმდევრობაზე, მე-10 სტრიქონში. სიგანეში ძებნის ხეები შეიძლება განსხვავდებოდეს, მაგრამ მანძილი d, რომელსაც ითვლის ალგორითმი, არ არის დამოკიდებული წვეროთა განხილვის რიგზე.
განვსაზღვროთ პროცედურის მუშაობის დრო. ყოველი წვერო რიგში თავსდება მხოლოდ ერთხელ და ასევე ერთხელ ხდება მისი ამოღება რიგიდან, ამიტომ რიგთან დაკავშირებულ ოპერაციებზე დაიხარჯება O(V) დრო. მოსაზღვრე წვეროთა სია ასევე ერთხელ განიხილება, როცა შესაბამისი წვერო ამოიღება რიგიდან. მოსაზღვრე წვეროთა სიებში ელემენტთა ჯამური სიგრძე კი E-ს ტოლია (არაორიენტირებულ გრაფში $2|E|$), ამიტომ ამ ოპერაციაზე დაიხარჯება O(E) დრო. ინიციალიზაციას სჭირდება O(V) დრო. მაშასადამე, ალგორითმის მუშაობის საერთო დრო იქნება O(V+E).
სიგანეში ძებნა პოულობს მანძილებს ფესვიდან გრაფის თითოეულ მიღწევად წვერომდე. განვსაზღვროთ **უმოკლესი გზის სიგრძე** (shortest-path distance) δ(s,v) - s ფესვიდან v წვერომდე, როგორც წიბოების მინიმალური რაოდენობა s ფესვიდან v წვერომდე რაიმე გზაზე. თუ გზა s-დან v-მდე არ არსებობს, მაშინ $\delta(s, v) = \infty$. $\delta(s, v)$ სიგრძის გზას s ფესვიდან v წვერომდე ეწოდება **უმოკლესი გზა** (shortest path). ასეთი გზა შეიძლება რამდენიმე იყოს. ქვემოთ განხილული იქნება უმოკლესი გზების უფრო ზოგადი სახე, როცა საქმე გვაქვს წონიან წიბოებთან და გზის სიგრძე წიბოების წონათა ჯამის ტოლია. ჩვენს შემთხვევაში კი წიბოთა წონები ერთეულის ტოლად ითვლება და გზის სიგრძე წიბოთა რაოდენობას უდრის.

ლემა 2.1. ვთქვათ მოცემულია G=(V,E) გრაფი (ორიენტირებული ან არაორიენტირებული) და მისი ნებისმიერი s წვერო. მაშინ მისი ნებისმიერი $(u, v) \in E$ წიბოსთვის სამართლიანია $\delta(s, v) \leq \delta(s, u) + 1$.

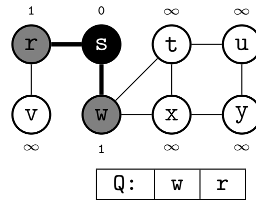
Proof. თუ u წვერო მიღწევადია s-დან, მაშინ მიღწევადია v-ც. ამ შემთხვევაში, უმოკლესი გზა s-დან v-დე არ შეიძლება იყოს იმაზე გრძელი, ვიდრე უმოკლესი გზა s-დან u-მდე, რომელსაც მოსდევს (u,v) წიბო. ასე, რომ დასამტკიცებელი უტოლობა სრულდება. ხოლო, თუ u წვერო არ არის მიღწევადი s-დან, მაშინ, $\delta(s, u) = \infty$ და უტოლობა ამ შემთხვევაშიც სრულდება. □

ლემა 2.2. ვთქვათ მოცემულია G=(V,E) გრაფი (ორიენტირებული ან არაორიენტირებული) და ვთქვათ, სრულდება პროცედურა BFS(G,s), მაშინ პროცედურის დასრულების შემდეგ, ყოველი $v \in V$ წვეროსთვის, მნიშვნელობა d[v], გამოთვლილი BFS(G,s) პროცედურით, აკმაყოფილებს უტოლობას $d[v] \geq \delta(s, v)$.

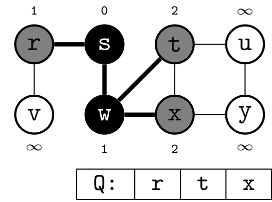
Proof. გამოვიყენოთ ინდუქცია ENQUEUE ოპერაციის რაოდენობის მიხედვით. ინდუქციის ჰიპოთეზა მდგომარეობს იმაში, რომ ყოველი $v \in V$ წვეროსთვის, სრულდება პირობა $d[v] \geq \delta(s, v)$.



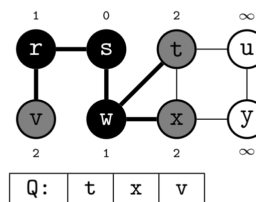
(a)



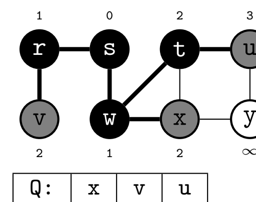
(b)



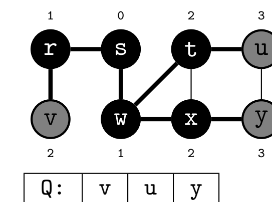
(c)



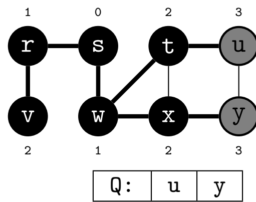
(d)



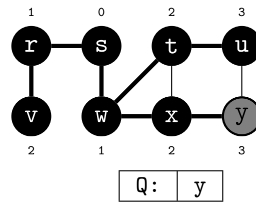
(e)



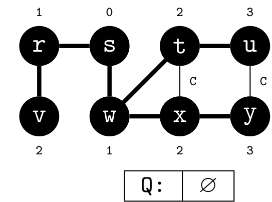
(f)



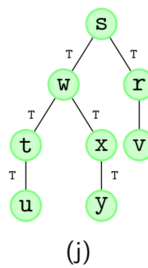
(g)



(h)



(i)



(j)

ნახ. 2.1:

ინდუქციის ბაზისი არის მდგომარეობა, რომელიც დგება s საწყისი წვეროს რიგში დამატების შემდეგ BFS(G,s) პროცედურის მე-7 სტრიქონში. ამ შემთხვევაში პირობა სრულდება: $d[s] = 0 = \delta(s, s)$, ხოლო $\forall v \in V - \{s\}$-სთვის $d[v] = \infty \geq \delta(s, v)$.
ინდუქციის ყოველ ბიჯზე განვიხილოთ თეთრი წვერო v, რომელიც იხსნება ძებნის პროცესში u წვეროდან. ინდუქციის ჰიპოთეზის თანახმად, $d[u] \geq \delta(s, u)$. მე-13 სტრიქონში მინიჭებისა და ლემა 2.1-ს გამოყენებით, მივიღებთ:

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$$

ამის შემდეგ, v წვერო ემატება რიგს. რადგან იგი ამ დროს იღებება რუხად, ხოლო 12-15 სტრიქონები სრულდება მხოლოდ თეთრი წვეროებისთვის, v წვერო რიგს აღარ დაემატება, ამრიგად, მისი მნიშვნელობა d[v] აღარ შეიცვლება, ასე რომ ინდუქციის ჰიპოთეზა სრულდება. □

ლემა 2.3. ვთქვათ, BFS(G,s) პროცედურის შესრულების პროცესში, Q რიგში შედის წვეროები $< v_1, v_2, \ldots, v_r >$, სადაც v_1 - Q რიგის თავია, ხოლო v_r - Q რიგის ბოლო. მაშინ ყოველი i=1, 2, ..., r-1-სთვის სამართლიანია $d[v_r] \leq d[v_1] + 1$ და $d[v_i] \leq d[v_{i+1}]$.

Proof. დავამტკიცოთ ინდუქციით, Q რიგთან ჩატარებული ოპერაციების რაოდენობის მიხედვით. ინდუქციის ბაზისად ჩავთვალოთ მდგომარეობა, როცა რიგში არის ერთი s წვერო. ამ შემთხვევაში ლემის პირობა სრულდება. ინდუქციის ყოველ ბიჯზე უნდა დავამტკიცოთ, რომ ლემა სრულდება Q რიგში წვეროს როგორც ჩამატების, ისე ამოღების შემდეგაც.
თუ რიგიდან ვიღებთ მის თავს, v_1-ს, რიგის ახალი თავი ხდება v_2 (თუ რიგი ცარიელდება რაიმე წვეროს რიგიდან ამოღებით, ლემა სრულდება). ინდუქციის ჰიპოთეზის თანახმად, $d[v_1] \leq d[v_2]$, მაგრამ მაშინ მივიღებთ $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, ხოლო ყველა დანარჩენი უტოლობა რჩება უცვლელი. ამრიგად, ლემა სრულდება, როცა რიგის ახალი თავი ხდება v_2.
ჩავამატოთ რიგში წვერო (BFS პროცედურის მე-15 სტრიქონი), ის გახდება v_{r+1}, (რადგან Q რიგი შეიცავს $< v_1, v_2, \ldots, v_r >$ წვეროებს), ამ მომენტში, u წვერო, რომლის მოსაზღვრე წვეროთა სია განიხილებოდა, უკვე ამოღებულია რიგიდან და ინდუქციის ჰიპოთეზის თანახმად, რიგის ახალი v_1 თავისთვის, სრულდება უტოლობა: $d[v_1] \leq d[u]$. ამრიგად, $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$, გარდა ამისა, ინდუქციის ჰიპოთეზის თანახმად, $d[v_r] \leq d[u] + 1$, და $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$, ხოლო დანარჩენი უტოლობები რჩება უცვლელი. ამრიგად ლემა სრულდება რიგში ახალი წვეროს ჩამატების შემდეგაც. □

შედეგი 2.1. ვთქვათ, პროცედურა BFS(G,s) შესრულების პროცესში, Q რიგს ემატება ჯერ v_i და შემდეგ v_j წვეროები, მაშინ v_j წვეროს რიგში ჩამატების მომენტში, სრულდება უტოლობა: $d[v_i] \leq d[v_j]$.

თეორემა 2.1. (სიგანეში ძებნის კორექტულობა) ვთქვათ, მოცემულია G=(V,E) გრაფი (ორიენტირებული ან არაორიენტირებული) და ვთქვათ, სრულდება პროცედურა BFS(G,s) s ფესვის მქონე G=(V,E) გრაფისათვის. მაშინ მუშაობის პროცესში BFS(G,s) ხსნის s-დან მიღწევად ყველა $v \in V$ წვეროს, და პროცედურის დამთავრების შემდეგ, ყველა $v \in V$ წვეროსთვის, შესრულდება ტოლობა $d[v] = \delta(s, v)$. ამის გარდა, s-დან მიღწევადი ნებისმიერი $v \neq s$ თვის ერთ-ერთი უმოკლესი გზა s-დან v-მდე არის უმოკლესი გზა s-დან π[v]-მდე, რომელსაც მოსდევს წიბო $(\pi[v], v)$.

Proof. დავუშვათ საწინააღმდეგო. ვთქვათ, რომელიმე წვეროსთვის, d მნიშვნელობა არ უდრის უმოკლესი გზის სიგრძეს. ვთქვათ, v არის მინიმალური $\delta(s, v)$ სიგრძის მქონე წვერო, იმ წვეროებს შორის, რომლისთვისაც არ არის სწორად გამოთვლილი d მნიშვნელობა. ცხადია, $v \neq s$. **ლემა** 2.2 თანახმად, $d[v] \geq \delta(s, v)$, ამიტომ, ჩვენი დაშვების გამო, $d[v] > \delta(s, v)$. v წვერო მიღწევადი უნდა იყოს s-დან, რადგან წინააღმდეგ შემთხვევაში, $\delta(s, v) = \infty \geq d[v]$. ვთქვათ, u არის წვერო, რომელიც უშუალოდ წინ უძღვის v წვეროს s-დან v-მდე უმოკლეს გზაზე, ასე, რომ შესრულდება $\delta(s, v) = \delta(s, u) + 1$. რადგანაც $\delta(s, u) = \delta(s, v)$, ხოლო v არის მინიმალური $\delta(s, v)$ სიგრძის მქონე წვერო, რომლისთვისაც არ არის სწორად გამოთვლილი d მნიშვნელობა, ამიტომ $d[u] = \delta(s, u)$. საბოლოოდ, მივიღებთ:

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 \tag{2.1}$$

ახლა, განვიხილოთ მომენტი, როცა პროცედურა BFS(G,s) იღებს u წვეროს Q რიგიდან მე-9 სტრიქონში. ამ მომენტში v წვერო შეიძლება იყოს თეთრი, რუხი ან შავი. ვაჩვენოთ, რომ სამივე შემთხვევისთვის მივიღებთ წინააღმდეგობას (2.1)-თან.
თუ v წვერო თეთრია, მაშინ მე-13 სტრიქონში სრულდება მინიჭება d[v] = d[u] + 1, რომელიც ეწინააღმდეგება (2.1)-ს. თუ v წვერო შავია, მაშინ ის უკვე ამოღებულია რიგიდან და **შედეგი** 2.1-ს თანახმად, $d[v] \leq d[u]$, რაც, აგრეთვე, ეწინააღმდეგება (2.1)-ს. თუ v წვერო რუხია, მას ეს ფერი შეეძლო მიეღო რიგიდან ω წვეროს ამოშლის დროს. ω წვერო ამოშლილია u წვეროს ამოშლამდე და მისთვის სრულდება ტოლობა $d[v] = d[\omega] + 1$. **შედეგი** 2.1-დან კი გამომდინარეობს, რომ $d[\omega] \leq d[u]$, ამიტომ $d[v] \leq d[u] + 1$, რომელიც ასევე ეწინააღმდეგება (2.1)-ს.
ამრიგად, ყოველი $v \in V$ წვეროსთვის, სრულდება ტოლობა $d[v] = \delta(s, v)$. დამტკიცების დასასრულებლად შევნიშნოთ, რომ თუ π[v]=u, მაშინ d[v]=d[u]+1. ამიტომ, უმოკლესი გზა s-დან v-მდე შეგვიძლია მივიღოთ, თუ ვიპოვით უმოკლეს გზას s-დან π[v]-მდე და შემდეგ გავივლით $(\pi[v], v)$ წიბოზე. □

s ფესვის მქონე G=(V,E) გრაფისათვის, განვიხილოთ G_π **წინამორბედობის ქვეგრაფი** (predecessor subgraph), როგორც $G_\pi = (V_\pi, E_\pi)$, სადაც:

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$$

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$$

G_π ქვეგრაფი წარმოადგენს **სიგანეში ძებნის ხეს** (breadth-first tree), თუ V_π შედგება s-დან მიღწევადი წვეროებისგან და ყოველი $v \in V_\pi$ წვეროსათვის G_π-ში არსებობს ერთადერთი მარტივი გზა s-დან v-ში, რომელიც ერთდროულად არის უმოკლესი გზა s-დან v-ში G გრაფში. სიგანეში ძებნის ხე წარმოადგენს ხეს, რადგან ის ბმულია და $|E_\pi| = |V_\pi| - 1$.
შემდეგი ლემა აჩვენებს, რომ s ფესვის მქონე G=(V,E) გრაფისათვის BFS(G,s) პროცედურის შესრულების შემდეგ, წინამორბედობის ქვეგრაფი წარმოადგენს სიგანეში ძებნის ხეს.

ლემა 2.4. G=(V,E) გრაფისათვის BFS(G,s) პროცედურის შესრულების შემდეგ, პროცედურა π-ს აგებს ისე, რომ წინამორბედობის ქვეგრაფი $G_\pi = (V_\pi, E_\pi)$ წარმოადგენს სიგანეში ძებნის ხეს.

Proof. მე-14 სტრიქონში π[v]=u მინიჭება ხორციელდება მაშინ და მხოლოდ მაშინ, როცა $(u, v) \in E$ და $\delta(s, v) < \infty$, ანუ როცა v მიღწევადია s-დან. ამიტომ, V_π შედგება V-ს იმ წვეროებისგან, რომლებიც მიღწევადია s-დან. რადგან G_π წარმოადგენს ხეს, **თეორემა** 1.1-ს მიხედვით, ის შეიცავს ერთადერთ გზას s-დან V_π სიმრავლის ყოველ წვერომდე. **თეორემა** 2.1-ს გამოყენებით, კი დავასკვნით, რომ ყოველი ასეთი გზა უმოკლესია. □

შემდეგი პროცედურა ბეჭდავს s-დან v-მდე ყველა წვეროს, მას შემდეგ, რაც BFS(G,s) პროცედურით უკვე აგებულია სიგანეში ძებნის ხე.

Algorithm 2: PRINT PATH

Input: გრაფი $G = (V, E)$, წვეროები s და v, მშობლების მასივი π
Output: გზა s-დან v-ში

```
1 PRINT-PATH(v) :
2     if v == s :
3         print(s);
4     elif π[v] == NIL :
5         print(' გზა არ არსებობს');
6     else:
7         PRINT-PATH(π[v]);
8         print(v);
```

ფუნქციის მუშაობის დრო დასაბეჭდი გზის სიგრძის პროპორციულია, რადგან ყოველი რეკურსიული გამოძახება ერთი ერთეულით ამცირებს გზას ფესვამდე.

2.2 სიღრმეში ძებნის ალგორითმი

სიღრმეში ძებნის ალგორითმი (depth-first search) იწყებს გრაფის წვეროების შემოვლას ნებისმიერი წვეროდან, ყოველ იტერაციაზე ალგორითმი გადადის მიმდინარე წვეროს მოსაზღვრე წვეროზე და ეს პროცესი გრძელდება მანამ, სანამ არ მიაღწევს ჩიხს, ანუ ისეთ მდგომარეობას, როცა წვეროს არ ჰყავს გაუვლელი მოსაზღვრე წვერო. ამ შემთხვევაში, ალგორითმი ბრუნდება ერთი წიბოთი უკან, წვეროში, რომლიდანაც ის მოხვდა ჩიხში და აგრძელებს პროცესს იმ ადგილიდან. ალგორითმი ასრულებს მუშაობას, როცა ბრუნდება საწყის წვეროში, რომელიც ამ მომენტისთვის უკვე ჩიხია. თუ გრაფში დარჩენილია გაუვლელი წვეროები, ალგორითმი ირჩევს ერთ-ერთ მათგანს და პროცესი მეორდება.
სიღრმეში ძებნის სტრატეგია შეიძლება ასეც ჩამოვაყალიბოთ: ვიაროთ გრაფში წინ (სიღრმეში), სანამ არსებობს გაუვლელი წიბო. თუ გაუვლელი წიბო აღარ არსებობს, დავბრუნდეთ უკან და ვეძებოთ სხვა გზა. ასე გავაგრძელოთ მანამ, სანამ არ ამოიწურება ყველა წვერო, რომელიც მიღწევადია საწყისიდან. თუკი გაუვლელი წვერო მაინც დარჩა, ავიღოთ ერთ-ერთი და გავიმეოროთ პროცესი, სანამ არ იქნება შემოვლილი გრაფის ყველა წვერო.
განივად ძებნის მსგავსად, როცა პირველად გამოვლინდება მოსაზღვრე v წვერო, u-ს მნიშვნელობა თავსდება $\pi[v]$-ში. მიიღება ხე (ან ხეები - თუკი ძებნა განმეორდება რამდენიმე წვეროდან). სიღრმეში ძებნის პროცესში მიიღება წინამორბედობის ქვეგრაფი, რომელიც ასე განისაზღვრება:

$$G_\pi = (V, E_\pi) \text{ , სადაც } E_\pi = \{(\pi[v], v) : v \in V \text{ და } \pi[v] \neq NIL\}$$

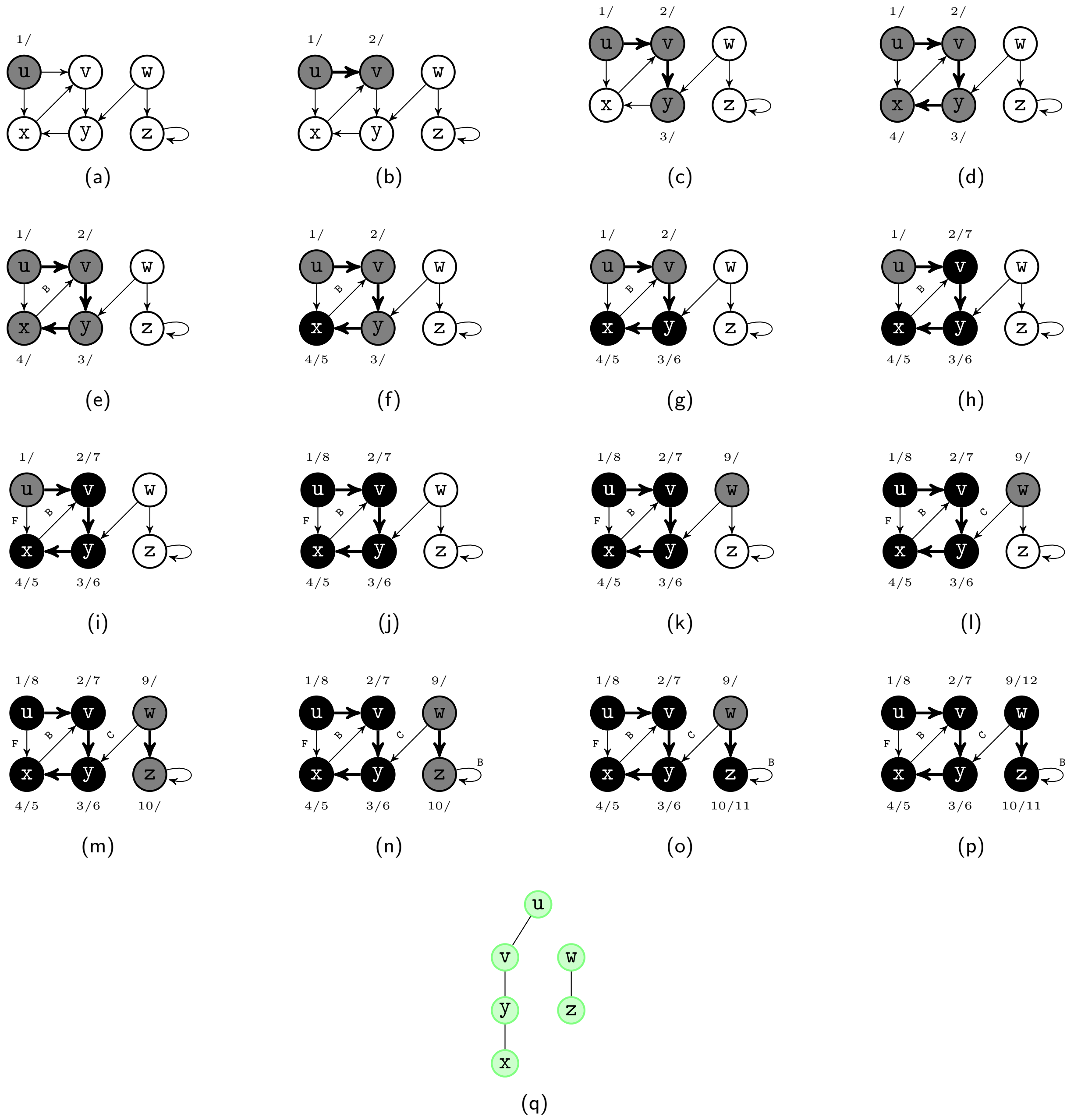
წინამორბედობის ქვეგრაფი წარმოადგენს სიღრმეში ძებნის ტყეს, რომელიც შეიძლება შედგებოდეს სიღრმეში ძებნის ხეებისაგან. E_π სიმრავლეში შემავალ წიბოებს უწოდებენ **ხის წიბოებს**.
სიღრმეში ძებნის ალგორითმიც იყენებს წვეროების შეფერვას. თავიდან ყველა წვერო თეთრია. წვეროს აღმოჩენის შემდეგ იგი ხდება რუხი. როცა წვერო მთლიანად დამუშავებულია, ანუ თუ მისი მოსაზღვრე წვეროების სია ბოლომდე განხილულია, იგი ხდება შავი. ყოველი წვერო ხვდება სიღრმეში ძებნის მხოლოდ ერთ ხეში (ამიტომ ეს ხეები არ გადაიკვეთებიან).
გარდა ამისა სიღრმეში ძებნა თითოეულ წვეროს მიუწერს დროის **ჭდეებს** (timestamp). ყოველ წვეროს აქვს ორი ჭდე: $d[u]$-ში ჩაიწერება, თუ როდის იქნა აღმოჩენილი წვერო (და გახდა რუხი), ხოლო $f[u]$-ში - როდის დასრულდა წვეროს დამუშავება (და გახდა შავი).
შეიძლება დროის ჭდეების ასეთი ინტერპრეტაციაც: მოვათავსოთ წვერო სტეკში, მისი აღმოჩენის მომენტში, და ამოვიღოთ სტეკიდან მაშინ როცა ის ხდება ჩიხი.
ქვემოთ მოყვანილ DFS პროცედურაში $d[u]$ და $f[u]$ წარმოადგენენ მთელ რიცხვებს 1-დან $2|V|$-მდე. ნებისმიერი წვეროსათვის სრულდება უტოლობა $d[u] < f[u]$. u წვერო იქნება თეთრი $d[u]$ მომენტამდე, $d[u]$-სა და $f[u]$-ს შორის იქნება რუხი, ხოლო $f[u]$-ის შემდეგ შავი.
მიმდინარე დროის time ცვლადი გლობალურია და გამოიყენება დროის ჭდეებისათვის.

Algorithm 3: Depth First Search (DFS)

Input: გრაფი $G = (V, E)$
Output: გრაფის სიღრმეში ძებნისას აღმოჩენილი წვეროების მიმდევრობა

```
1  DFS(G) :
2      for ∀u ∈ V :
3          color[u] = TeTri;
4          π[u] = NIL;
5      time = 0;
6      for ∀u ∈ V :
7          if color[u] == TeTri :
8              DFS-VISIT(u)
9      return d, f, π
10 DFS-VISIT(u) :
11     color[u] = ruxi;
12     time++; d[u] = time;
13     for ∀v ∈ Adj[u] :
14         if color[v] == TeTri :
15             π[v] = u;
16             DFS-VISIT(v);
17     color[u] = Savi;
18     time++; f[u] = time;
```

2-4 სტრიქონებში ყველა წვერო ხდება თეთრი, ხოლო π-ს მიენიჭება მნიშვნელობა NIL. მე-5-ე სტრიქონში განისაზღვრება საწყისი (ნულოვანი) დრო. 6-8 სტრიქონებში ხდება პროცედურა DFS-VISIT-ს გამოძახება იმ წვეროებისათვის, რომლებიც თეთრია გამოძახების მომენტისათვის (პროცედურის წინა გამოძახებამ ისინი შეიძლება შავი გახადოს). ეს წვეროები წარმოადგენენ სიღრმეში ძებნის ხეთა ფესვებს.
DFS-VISIT(u) პროცედურის გამოძახებისას u წვერო თეთრია. მე-11 სტრიქონში ის რუხი ხდება. მე-12-ე სტრიქონში მისი აღმოჩენის დრო შეიტანება $d[u]$-ში (წინასწარ დროის მთვლელი 1-ით იზრდება). 13-16 სტრიქონებში განიხილება u-ს მოსაზღვრე წვეროები და ხდება DFS-VISIT პროცედურის გამოძახება u-ს მოსაზღვრე იმ წვეროებისათვის, რომლებიც თეთრი ფერის არიან გამოძახების მომენტში. u-ს ყველა მოსაზღვრე წვეროს განხილვის შემდეგ, იგი ხდება შავი და $f[u]$-ში იწერება ამ მოვლენის ამსახველი დრო (სტრ. 17, 18).
სურ. 2.2-ზე მოცემულია DFS პროცედურის შესრულების პროცესი და სიღრმეში ძებნის ტყე ორიენტირებული გრაფისათვის. მუქად აღნიშნულია **ხის** წიბოები (tree edges). წიბოები, რომლებიც არ შედიან სიღრმეში ძებნის ტყეში, აღნიშნულნი არიან შემდეგნაირად: **უკუწიბოები** - B (back edges), **ჯვარედინი** წიბოები - C (cross edges), **პირდაპირი** წიბოები - F (forward edges).
შევნიშნოთ, რომ სიღრმეში ძებნის შედეგი შეიძლება დამოკიდებული იყოს წვეროების განხილვის თანმიმდევრობაზე, (DFS პროცედურის მე-6 სტრ.), აგრეთვე, მოსაზღვრე წვეროების განხილვის თანმიმდევრობაზე, (DFS-VISIT პროცედურის მე-13 სტრ.). პრაქტიკაში აღნიშნული გარემოება არ ქმნის პრობლემას, რადგან სიღრმეში ძებნის ალგორითმის ნებისმიერი შედეგი შეიძლება ეფექტურად იქნას გამოყენებული და ფაქტობრივად ერთნაირ შედეგებს იძლევა სიღრმეში ძებნაზე დაფუძნებული ალგორითმებისთვის.



ნახ. 2.2:

დავითვალოთ DFS პროცედურის მუშაობის დრო: ციკლები 2-4 და 6-8 სტრიქონებში მოითხოვენ $\Theta(V)$ დროს. პროცედურა DFS-VISIT-ს გამოძახება ხდება მხოლოდ ერთხელ ყოველი წვეროსთვის. DFS-VISIT(u) პროცედურის გამოძახებისას ციკლი 13-16 სტრიქონებში სრულდება $|Adj[v]|$-ჯერ. და რადგან

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

ამიტომ DFS-VISIT პროცედურის 13-16 სტრიქონების შესრულების დრო იქნება $\Theta(E)$. DFS პროცედურის მუშაობის დრო კი - $\Theta(V+E)$.
შევნიშნოთ, რომ სიღრმეში ძებნის ხეებისაგან შედგენილი წინამორბედობის ქვეგრაფი ზუსტად შეესაბამება DFS-VISIT პროცედურის რეკურსიული გამოძახებების სტრუქტურას. სახელდობრ, $u = \pi[v]$ მაშინ და მხოლოდ მაშინ, როცა მოხდა DFS-VISIT(v) გამოძახება წვეროს მოსაზღვრე წვეროების განხილვის დროს.
სიღრმეში ძებნის მეორე მნიშვნელოვანი თვისება იმაში მდგომარეობს, რომ წვეროთა აღმოჩენისა და დამუშავების დამთავრების დროები ქმნიან წესიერ ფრჩხილურ სტრუქტურას. აღვნიშნოთ u წვეროს პოვნა მარცხენა ფრჩხილითა და წვეროს სახელით - "$(u$", ხოლო მისი დამუშავების დამთავრება წვეროს სახელითა და მარჯვენა ფრჩხილით - "$u)$". მოვლენათა ჩამონათვალი იქნება ფრჩხილებისაგან წესიერად აგებული გამოსახულება. მაგალითად სურ. 2.3-ზე გამოსახული გრაფისათვის:

$$(s(z(y(xx)y)(ww)z)s)(t(vv)(uu)t)$$

ნახ. 2.3:

ადგილი აქვს მტკიცებულებებს:

თეორემა 2.2. (ფრჩხილური სტრუქტურის შესახებ) სიღრმეში ძებნის დროს ორიენტირებულ ან არაორიენტირებულ $G = (V, E)$ გრაფში ნებისმიერი ორი u და v წვეროსათვის სრულდება მხოლოდ ერთ-ერთი დებულება შემდეგი სამიდან:

- $[d[u], f[u]]$ და $[d[v], f[v]]$ მონაკვეთები არ გადაიკვეთება
- $[d[u], f[u]]$ მონაკვეთი მთლიანად შედის $[d[v], f[v]]$ შიგნით და u წვერო v-ს შთამომავალია სიღრმეში ძებნის ხეში
- $[d[v], f[v]]$ მონაკვეთი მთლიანად შედის $[d[u], f[u]]$ შიგნით და v წვერო u-ს შთამომავალია სიღრმეში ძებნის ხეში

შედეგი 2.2. v წვერო u-ს შთამომავალია სიღრმეში ძებნის ტყეში ორიენტირებულ ან არაორიენტირებულ $G = (V, E)$ გრაფში მაშინ და მხოლოდ მაშინ, როცა $d[u] < d[v] < f[v] < f[u]$.

თეორემა 2.3. (თეთრი გზის შესახებ). v წვერო u-ს შთამომავალია სიღრმეში ძებნის ტყეში (ორიენტირებულ ან არაორიენტირებულ) $G = (V, E)$ გრაფში მაშინ და მხოლოდ მაშინ, როცა $d[u]$ დროის მომენტში (u წვეროს აღმოჩენის), არსებობს გზა u-დან v-ში, რომელიც შედგება მხოლოდ თეთრი წვეროებისაგან.

Proof. დავუშვათ, v წვერო u-ს შთამომავალია. ვთქვათ, w ნებისმიერი წვეროა u-დან v-ში მიმავალ გზაზე სიღრმეში ძებნის ტყეში, ასე რომ w წარმოადგენს u-ს შთამომავალს. **შედეგი** 2.2-ს თანახმად, $d[u] < d[w]$, ამიტომ $d[u]$ მომენტში, w წვერო თეთრია.
ახლა, დავუშვათ, რომ $d[u]$ მომენტში, არსებობს გზა u-დან v-ში, რომელიც შედგება მხოლოდ თეთრი წვეროებისაგან, მაგრამ v წვერო არ ხდება u-ს შთამომავალი სიღრმეში ძებნის ხეში. ზოგადობის შეუზღუდავად შეგვიძლია ვიგულისხმოთ, რომ u-დან v-მდე გზის ყველა დანარჩენი წვერო ხდება u-ს შთამომავალი (წინააღმდეგ შემთხვევაში, v-ს როლში ავიღებთ აღნიშნულ გზაზე u-სთან უახლოეს წვეროს, რომელიც არ ხდება u-ს შთამომავალი). ვთქვათ, w არის v წვეროს წინამორბედი ამ გზაზე, ე.ი. ის u-ს შთამომავალია. (u და w შეიძლება სინამდვილეში ერთი წვერო იყოს). **შედეგი** 2.2-ის თანახმად, $f[w] \leq f[u]$. შევნიშნოთ, რომ v წვეროს აღმოჩენა

ხდება მას შემდეგ, რაც აღმოჩენილია u, მაგრამ w წვეროს დამუშავების დამთავრებამდე. ამრიგად, $d[u] < d[v] < f[w] < f[u]$. **თეორემა** 2.2-ის თანახმად, $[d[v], f[v]]$ მონაკვეთი მთლიანად ეკუთვნის $[d[u], f[u]]$ მონაკვეთს. **შედეგი** 2.2-ის თანახმად კი, v წვერო გახდება u-ს შთამომავალი. □

გრაფის შემოვლის ალგორითმების გამოყენება

ბმული კომპონენტები

როგორც ვიცით, არაა აუცილებელი, რომ გრაფში ყველა ნაწილი ბმული იყოს, ანუ შეიძლება არსებობდეს ორი წვერო, რომელთა შორის შემაერთებელი გზა არ მოიძებნება. ასეთი ცალკეული კომპონენტების პოვნა ფუნდამენტური ამოცანაა გრაფთა თეორიაში: როდესაც რაიმე ამოცანა დაყოფილ გრაფებზე უნდა ამოიხსნას, უმეტეს შემთხვევაში უნდა დავადგინოთ დამოუკიდებელი ნაწილები და ისინი ცალ-ცალკე დავამუშაოთ.
მაგალითად, თუ მოცემულია რაიმე სიმრავლე, მასზე მოცემული ექვივალენტობის მიმართება, როგორც ვიცით, ამ სიმრავლეს ექვივალენტურობის კლასებად ყოფს და თუ ამ მიმართებას გრაფის სახით გამოვხატავთ, თითო კლასი ამ გრაფის ბმულობის კომპონენტი იქნება.

როგორც სიღრმეში, ასევე სიგანეში ძებნის ალგორითმების გამოყენებით ადვილად შეიძლება გრაფის ბმულობის კომპონენტების გამოყოფა: ავირჩევთ ნებისმიერ წვეროს და ამ რომელიმე ალგორითმით შემოვივლით დანარჩენ შესაძლო წვეროებს, რომლებსაც ერთი და იგივე ნიშანს დავადებთ (მაგალითად, მთვლელის რიცხვი). თუ გრაფში სხვა წვეროებიც დარჩა, მთვლელს ერთით გავზრდით და იგივე პროცედურას გავიმეორებთ მანამ, სანამ გრაფის ყველა წვეროს რაიმე ნიშანი არ დაედება.

სავარჯიშო 2.1: დაწერეთ ბმულობის კომპონენტების გამოყოფის ალგორითმი, დაამტკიცეთ მისი სისწორე და გამოითვალეთ ბიჯების ზედა ზღვარი.

ხეებისა და ციკლების პოვნა

ხეები - აციკლური (უციკლო) გრაფები - საინტერესო გრაფთა ყველაზე მარტივ კლასს ქმნიან. სიღრმეში ძებნის მეთოდი პირდაპირ გვაძლევს იმის პასუხს, არის თუ არა მოცემული გრაფი ხე: თუ ძებნის პროცესში განვლილ წიბოს აღვნიშნავთ როგორც „ხის წიბოს", ხოლო ისეთ წიბოს, რომელსაც არ გადავივლით (მაგალითად ისეთს, რომელსაც ერთი წვეროდან უკვე შესწავლილ წვეროში მივყავართ) აღვნიშნავთ, როგორც „დამატებითს", მოცემული გრაფი იქნება ხე მაშინ და მხოლოდ მაშინ, თუ სიღრმეში ძებნის შემდეგ დამატებითი წიბოები არ აქვს. რადგან ნებისმიერი ხისათვის $|E| = |V| - 1$, ამ ალგორითმის დროის ზედა ზღვარი იქნება $O(|V|)$.
თუ გრაფი შეიცავს ციკლს, მისი აღმოჩენა შეიძლება პირველივე დამატებითი წიბოს პოვნით: თუ აღმოჩნდა დამატებითი წიბო (u, v), მაშინ უკვე შექმნილ ხეში უნდა არსებობდეს გზა u წვეროდან v წვეროში და იგი (u, v) წიბოსთან ერთად მოგვცემს ციკლს.

სავარჯიშო 2.2: დაამტკიცეთ, რომ ნებისმიერი დამატებითი (u, v) წიბოს წვერეობს შორის არსებობს შემაერთებელი გზა.

სავარჯიშო 2.3: სიღრმეში ძებნის გამოყენებით დაწერეთ ალგორითმი, რომელიც მოცემული გრაფისათვის დაადგენს, არის თუ არა იგი ხე და თუ არა, ციკლებსაც იპოვნის.

სავარჯიშო 2.4: განიხილეთ წინა ამოცანაში დაწერილი ალგორითმი. შეიძლება თუ არა მისი საშუალებით ყველა ციკლის აღმოჩენა?

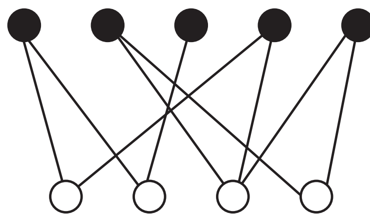
სავარჯიშო 2.5: შეიძლება თუ არა იგივე ამოცანის სიგანეში ძებნის მეთოდით გადაჭრა?

ორად შეღებილი გრაფები

გრაფის შეღებვის ზოგად ამოცანაში გრაფის წვეროები ისე უნდა შეიღებოს, რომ წიბოთი შეღებილ ორ წვეროს სხვადასხვა ფერი ქონდეს. ცხადია, თუ ყველა წვეროს სხვადასხვა ფრად შევღებავთ, ეს ამოცანა გადაიჭრება, მაგრამ საინტერესოა გრაფის რაც შეიძლება ცოტა ფრად შეღებვის ამოცანა - მოკლედ: გრაფის შეღებვის ამოცანა - რომელიც ფართოდ გამოიყენება ალგორითმების თეორიასა თუ პრაქტიკაში:

- ეფექტური ცხრილების შედგენაში - მაგალითად, მრავალბირთვიან პროცესორებში ამოცანის ნაწილების პარალელურად დამუშავების მიზნით - რა ნაწილი რის შემდეგ უნდა დამუშავდეს;
- რადიოტალღების სიხშირეების ეფექტურ დადგენაში - მაგალითად, თუ ორი მომხმარებელი რადიოგადამცემს ხმარობს, ახლოს მდგომებს სხვადასხვა სიხშირეები უნდა ქონდეთ, შორს მდგომებს შეიძლება ერთი და იგივე;
- რეგისტრების ეფექტურად დანაწილების ამოცანა - პროცესორის რეგისტრების გამოყენებით მონაცემების დამუშავება გაცილებით უფრო სწრაფად შეიძლება, ვიდრე RAM მეხსიერებიდან. მაგრამ რადგან ხშირად ცვლადთა რაოდენობა რეგისტრების რაოდენობას აჭარბებს, საჭიროა იმის დადგენა, რა დროს რომელი ცვლადი ჩაიწეროს რეგისტრებში;
- სახეთა ამოცნობაში - მაგალითად, მოცემული სურათით კატალოგში ადამიანის მოძებნა;
- არქეოლოგიური ან ბიოლოგიური მასალის ანალიზი - როგორც ბიოლოგიაში, ასევე არქეოლოგიაში მონაცემები ხის სახით შეგვიძლია შევინახოთ: ერთი სახეობა ან კულტურა მეორედან მომდინარეობს, ერთი სახეობა ან კულტურა სხვა რამოდენიმეს წარმოშობს.

ზოგადად, გრაფის მინიმალურად შეღებვის ამოცანა (ან მისი ***ქრომატული რიცხვის*** დადგენის ამოცანა, როგორც მას უწოდებენ ხოლმე), ძნელი გადასაჭრელია. მისი ერთ-ერთი მნიშვნელოვანი ქვეამოცანაა, შეიძლება თუ არა მოცემული გრაფის ორ ფრად შეღებვა, ან, სხვა სიტყვებით რომ ვთქვათ, ისეთ ორ ნაწილად დაყოფა, რომელშიც წვეროები ერთმანეთისგან იზოლირებულნი არიან. ასეთი გრაფის მაგალითია მოყვანილი ნახაზში 2.4.



ნახ. 2.4: ორად შეღებილი გრაფის მაგალითი

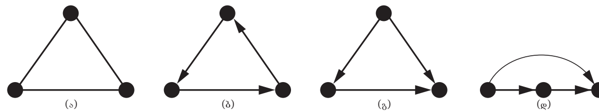
ორად შეღებილ გრაფს ზოგჯერ *ორად დაყოფილსაც* უწოდებენ. იმის დასადგენად, შეიძლება თუ არა მოცემული გრაფის ორად დაყოფა, შემდეგი სტრატეგიით შეიძლება მოქმედება:
ვირჩევთ ერთ-ერთ წვეროს, რომელსაც ვღებავთ რომელიმე ფრად (დავუშვათ, თეთრად). სიღრმეში ან სიგანეში ძებნით ახლად აღმოჩენილ წვეროს ვღებავთ მისი მშობლის (იმ წვეროსი, რომელთანაც არის დაკავშირებული) განსხვავებული ფერით (თეთრი ან შავი). შემდეგ *ყოველი აღმოჩენილი* წვეროსათვის ვამოწმებთ, არის თუ არა იგი მიერთებული ორ ერთსა და იმავე ფრად შეღებილ წვეროსთან და თუ ასეა, ეს იმას უნდა ნიშნავდეს, რომ გრაფი არ შეიღებება (არ დაიყოფა) ორად. თუ ალგორითმი ამ სახის კონფლიქტის გარეშე დასრულდება, ეს იმას უნდა ნიშნავდეს, რომ გრაფი ორად შეიღება.

სავარჯიშო 2.6: დაამტკიცეთ ამ მეთოდის სისწორე. დამოკიდებულია თუ არა ეს მეთოდი იმაზე, თუ რომელ საწყის წვეროს ავიღებთ?

სავარჯიშო 2.7: ამ მეთოდზე დაყრდნობით დაწერეთ ალგორითმი, რომელიც დაადგენს, შეიძლება თუ არა გრაფის ორად შეღებვა.

ტოპოლოგიური დალაგება

განვიხილოთ ნახ. 2.5-ში მოყვანილი სამი გრაფის მაგალითი.
პირველი გრაფი არაა მიმართული და შეიცავს ციკლს; მეორე მიმართულია და ციკლს შეიცავს, ხოლო მესამე კი მიმართულია, მაგრამ ციკლს არ შეიცავს (აციკლურია), რადგან ვერ მოვძებნით ისეთ *დაშვებულ* გზას, რომელიც გაყვება ისრებს და რომელიმე წვეროდან ისევ იგივე წვეროში დაგვაბრუნებდა.



ნახ. 2.5: არამიმართული ციკლური, მიმართული ციკლური და მიმართული აციკლური გრაფები

ასეთ სტრუქტურას *აციკლური მიმართული გრაფი* ეწოდება და მათი დახაზვა შეიძლება ისე, რომ ყველა წიბო *მარცხნიდან მარჯვნივ* იყოს მიმართული (მაგალითად, ნახ. 2.5(დ)), რასაც ამ გრაფის ***ტოპოლოგიურ დალაგებას*** უწოდებენ.

აღსანიშნავია, რომ მხოლოდ აციკლურ მიმართულ გრაფებს შეიძლება მოვუძებნოთ ტოპოლოგიური დალაგება, რადგან ნებისმიერი ციკლი რომელიმე კვანძიდან *უკან* (ანუ მარჯვნიდან მარცხნივ) დაგვაბრუნებდა, თანაც აციკლურ მიმართულ გრაფებს *ყოველთვის* მოვუძებნით ერთ ტოპოლოგიურ დალაგებას მაინც.

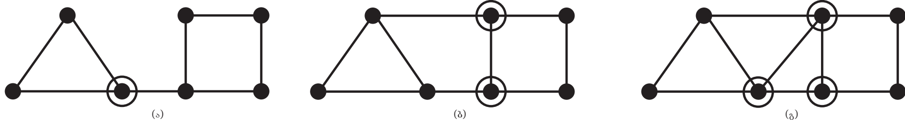
ტოპოლოგიურ დალაგებას დიდი მნიშვნელობა აქვს მთელ რიგ პრაქტიკულ ამოცანებში, მაგალითად ზემოთ ნახსენებ ცხრილის შედგენაში.

შიღრმეში ძებნის ალგორითმით ადვილად შეგვიძლია მიმართული გრაფის აციკლურობის დადგენა (რაც დამოკიდებულია იმაზე, შევხვდება თუ არა მუშაობის პროცესში ზემოთ ნახსენები „დამატებითი წიბოები"). თუ გრაფი აციკლურია, მაშინ ალგორითმის მსვლელობისას შექმნილი R მიმდევრობა ტოპოლოგიური დალაგების თანმიმდევრობას გვაძლევს.

სავარჯიშო 2.8: დაამტკიცეთ, რომ სიღრმეში ძებნის პროცესში შექმნილი R მიმდევრობა ტოპოლოგიური დალაგების თანმიმდევრობას გვაძლევს.

საარტიკულაციო კვანძები

ხშირად საჭიროა იმის დადგენა, თუ მინიმუმ რამდენი კვანზის ამოგდებაა საჭირო ბმული გრაფიდან იმისათვის, რომ იგი ნაწილებად დაიშალოს. ქვემოთ მოყვანილ ნახაზში ნაჩვენებია სამი გრაფი, რომელთა დაშლა *მინიმუმ* ერთი (ა), ორი (ბ) და სამი (გ) კვანძის ამოგდებით შეიძლება. ამ შემთხვევაში იტყვიან, რომ გრაფია *ერთად* ბმული, ან *ორად* ბმული, *სამად* ბმული და, ზოგადად, n-ად ბმული.
იტყვიან ასევე, რომ გრაფის ბმულობის კოეფიციენტია n.



ნახ. 2.6: ერთად, ორად და სამად ბმული გრაფი

თუ გრაფის ბმულობის კოეფიციენტია 1, მაშინ იტყვიან, რომ მას აქვს ე.წ. ***საარტიკულაციო კვანძი.***

საარტიკულაციო კვანძების ძებნა ძალიან მნიშვნელოვანია მაგალითად საკომუნიკაციო ქსელების სტაბილურობის დადგენაში. ზოგადად, რაც უფრო მაღალია ქსელის შესაბამისი გრაფის ბმულობის კოეფიციენტი, მით უფრო სტაბილურია იგი.
ადვილი შესამჩნევია, რომ საარტიკულაციო კვანძების ძებნა გრაფიდან რიგ-რიგობით წვეროების ამოგდებითა და დარჩენილი სტრუქტურის ბმულობაზე შემოწმებით შეიძლება.

სავარჯიშო 2.9: დაწერეთ ალგორითმი, რომლითაც გრაფის საარტიკულაციო კვანძების არსებობას დავადგენთ. დაამტკიცეთ მისი სისწორე და დაითვალეთ ბიჯების ზედა ზღვარი.

2.3 წიბოთა კლასიფიკაცია

გრაფის წიბოები იყოფა რამდენიმე კატეგორიად იმის მიხედვით, თუ რა როლს თამაშობენ ისინი სიღრმეში ძებნის დროს. ეს კლასიფიკაცია სასარგებლოა სხვადასხვა ამოცანების განხილვისას. მაგალითად, ორიენტირებულ გრაფს არ გააჩნია ციკლები მაშინ და მხოლოდ მაშინ, როცა სიღრმეში ძებნის ალგორითმი ვერ პოულობს მასში "უკუწიბოებს".
G გრაფისთვის G_π სიღრმეში ძებნის ტყის გამოყენებით შეიძლება განვსაზღვროთ წიბოების ოთხი ტიპი:

1. **ხის წიბოები** (tree edges) - ესაა G_π გრაფის წიბოები. (u, v) წიბო იქნება ხის წიბო, თუ v წვერო აღმოჩენილია ამ წიბოს დამუშავების დროს.

2. **უკუწიბოები** (back edges) - ესაა (u, v) წიბოები, რომლებიც აერთებენ u წვეროს მის v წინაპართან სიღრმეში ძებნის ხეზე. ორიენტირებული გრაფებისათვის დამახასიათებელი მარყუჟები უკუწიბოებად ითვლებიან.

3. **პირდაპირი წიბოები** (forward edges) - ესაა (u, v) წიბოები, რომლებიც აერთებენ წვეროს მის შთამომავალთან, მაგრამ არ შედიან სიღრმეში ძებნის ხეში.

4. **ჯვარედინი წიბოები** (cross edges) - გრაფის ყველა სხვა წიბო. მათ შეუძლიათ შეაერთონ სიღრმეში ძებნის ხის ორი ისეთი წვერო, რომელთა შორის არც ერთი არაა მეორის წინაპარი ან წვეროები, რომლებიც სხვადასხვა ხეებს ეკუთვნიან.

2. (ბ) ნახაზზე გრაფი მოცემულია იმდაგვარად, რომ ხის წიბოები და პირდაპირი წიბოები მიემართებიან ქვემოთ, ხოლო უკუწიბოები - ზემოთ.
DFS ალგორითმით შესაძლებელია წიბოთა კლასიფიკაცია. (u, v) წიბოს ტიპი შეიძლება განისაზღვროს v წვეროს ფერით, პირველი დამუშავების დროს (არ ხერხდება მხოლოდ პირდაპირ და ჯვარედინ წიბოებს შორის განსხვავების პოვნა):

1. თეთრი ფერი - ხის წიბო

2. რუხი ფერი - უკუწიბო

3. შავი - პირდაპირი ან ჯვარედინი წიბო

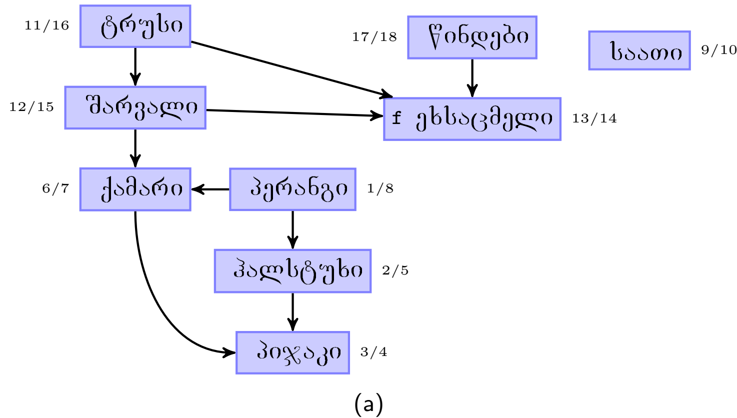
პირველი შემთხვევა უშუალოდ ალგორითმის მუშაობიდან გამომდინარეობს. მეორე შემთხვევისთვის შევნიშნოთ, რომ რუხი წვეროები ყოველთვის ქმნიან შთამომავლების მიმდევრობას, რომლებიც შეესაბამებიან DFS-VISIT პროცედურის გამოძახებებს. რუხი წვეროების რაოდენობა ერთით მეტია სიღრმეში ძებნის ხეზე უკანასკნელად გახსნილი წვეროს სიღრმეზე. წვეროების აღმოჩენა ხდება ამ მიმდევრობის პირველი, "ყველაზე ღრმა", რუხი წვეროდან, ასე რომ წიბო, რომელიც აღწევს სხვა რუხ წვეროს, აღწევს საწყისი წვეროს წინაპარს. მესამე შემთხვევისთვის, პირდაპირი და ჯვარედინი წიბოების განსასხვავებლად შეგვიძლია გამოვიყენოთ d-ს მნიშვნელობა: თუ $d[u] < d[v]$, მაშინ (u, v) წიბო პირდაპირია, ხოლო თუ $d[u] > d[v]$ - ჯვარედინი.
არაორიენტირებული გრაფი საჭიროებს განსაკუთრებულ განხილვას, რადგან ერთი და იგივე წიბო $(u, v) = (v, u)$ ორჯერ უნდა დამუშავდეს (თითოჯერ ორივე ბოლოდან) და შესაძლოა სხვადასხვა კატეგორიაში მოხვდეს. ამ შემთხვევაში ის უნდა მივაკუთვნოთ იმ კატეგორიას, რომელიც ჩვენს ჩამონათვალში უფრო წინ დგას. იმავე შედეგს მივიღებთ, თუკი ჩავთვლით, რომ წიბოს ტიპი განისაზღვრება მისი პირველი დამუშავებისას და არ იცვლება მეორე დამუშავებით. ირკვევა, რომ ასეთი შეთანხმებისას პირდაპირი და ჯვარედინი წიბოები არაორიენტირებულ გრაფში არ იქნება და ადგილი აქვს თეორემას.

თეორემა 2.4. სიღრმეში ძებნისას არაორიენტირებულ G გრაფში ნებისმიერი წიბო ან ხის წიბოა, ან უკუწიბო.

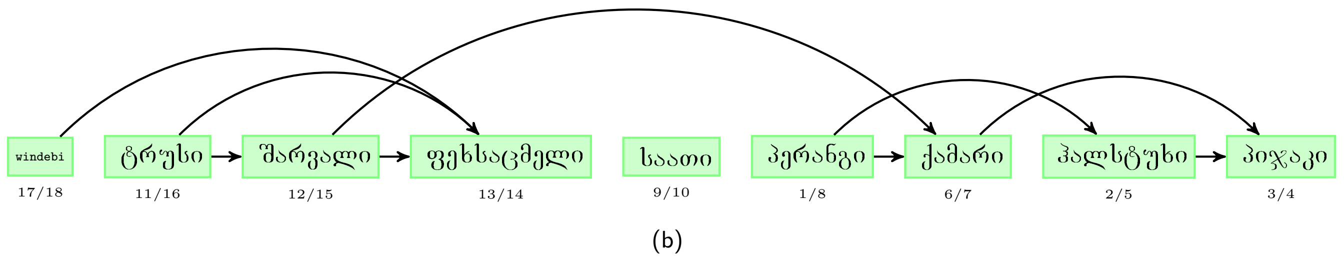
2.4 ტოპოლოგიური სორტირება

ვთქვათ, მოცემულია ორიენტირებული აციკლური გრაფი (directed acyclic graph). ამ გრაფის **ტოპოლოგიური სორტირება** (topological sort) გულისხმობს წვეროების განლაგებას ისეთი წრფივი თანმიმდევრობით, რომ ნებისმიერი წიბო მიმართული იყოს ამ თანმიმდევრობაში ნაკლები ნომრის მქონე წვეროდან მეტი ნომრის მქონე წვეროსაკენ. ცხადია, რომ თუკი გრაფი შეიცავს ციკლს, ასეთი თანმიმდევრობა არ იარსებებს. ამოცანა შეგვიძლია სხვაგვარადაც დავსვათ: განვალაგოთ გრაფის წვეროები ჰორიზონტალურ წრფეზე ისე, რომ ყველა წიბო მიმართული იყოს მარცხნიდან მარჯვნივ.
მაგალითისათვის განვიხილოთ ასეთი შემთხვევა: დაბნეული პროფესორისათვის დილემადაა ქცეული დილაობით ჩაცმა. მან ზოგიერთი რამის ჩაცმისას აუცილებლად უნდა დაიცვას მოქმედებების თანმიმდევრობა (მაგ.: ჯერ

წინდები, შემდეგ ფეხსაცმელი), ზოგ შემთხვევაში კი თანმიმდევრობას მნიშვნელობა არა აქვს (მაგ.: წინდები და შარვალი). სურ. 2.7ა-ზე ეს დამოკიდებულებანი მოცემულია ორიენტირებული გრაფის სახით. (u, v) წიბო აღნიშნავს, რომ u უნდა იქნას ჩაცმული v-ზე ადრე. ამ გრაფის ტოპოლოგიური სორტირების ერთ-ერთი ვარიანტი მოცემულია სურ. 2.7ბ-ზე.



(a)



(b)

ნახ. 2.7:

წვეროების გვერდით მითითებულია მათი დამუშავების დაწყებისა და დამთავრების დროები. სურ. 2.7ბ-ში გრაფი ტოპოლოგიურად სორტირებულია. წვეროები დალაგებულია დამუშავების დამთავრების დროთა მიხედვით. ყველა წიბო მიმართულია მარცხნიდან მარჯვნივ.
შემდეგი ალგორითმი ტოპოლოგიურად ალაგებს ორიენტირებულ აციკლურ გრაფს.

Algorithm 4: Topological Sort

Input: (DAG) ორიენტირებული აციკლური გრაფი $G = (V, E)$
Output: ტოპოლოგიურად სორტირებული წვეროების მიმდევრობა

```
1 თIჩ”-შლთ() :
2 |  L = [];      // ცარიელი სია
3 |  DFS(G);      // წვეროს დამუშავების დამთავრებისას
4 |               //(DFS-VISIT,  სტრ.18)
5 |               // დავამატოთ წვერო  სიის დასაწყისში
6 |  return L
```

ტოპოლოგიური სორტირება სრულდება $\Theta(V+E)$ დროში, რადგან ამდენი დრო სჭირდება სიღრმეში ძებნას, ხოლო სიაში წვეროს ჩაწერას სჭირდება $O(E)$ დრო. ალგორითმის სისწორე მტკიცდება შემდეგი ლემის დახმარებით:

ლემა 2.5. ორიენტირებული გრაფი არ შეიცავს ციკლებს მაშინ და მხოლოდ მაშინ, როცა სიღრმეში ძებნის ალგორითმი ვერ პოულობს მასში უკუწიბოებს.

Proof. ვთქვათ, არსებობს (u, v) უკუწიბო, მაშინ v არის u-ს წინაპარი სიღრმეში ძებნის ხეზე, ამრიგად, გრაფში არსებობს გზა v-დან u-ში და (u, v) წიბო ასრულებს ციკლს.
ვთქვათ, გრაფში გვაქვს ციკლი c. დავამტკიცოთ, რომ ამ შემთხვევაში სიღრმეში ძებნა აუცილებლად იპოვის უკუწიბოს. ციკლის წვეროებიდან ამოვირჩიოთ წვერო v, რომელიც პირველად იქნა აღმოჩენილი, და ვთქვათ, (u, v) c-ში შემავალი წიბოა. $d[v]$ მომენტში v-დან u-ში მივყავართ თეთრი წვეროებისაგან შემდგარ გზას. **თეორემა** 2.3 **(თეთრი გზის შესახებ)**-ის თანახმად, u გახდება სიღრმეში ძებნის ტყეში, ამიტომ (u, v) იქნება უკუწიბო. □

თეორემა 2.5. *TOPOLOGICAL-SORT*(G) პროცედურა სწორად ასრულებს ტოპოლოგიურ სორტირებას აციკლური ორიენტირებული G გრაფისთვის.

Proof. ვთქვათ, $G = (V, E)$ აციკლური ორიენტირებული გრაფისთვის შესრულდა DFS პროცედურა, რომელიც გამოითვლის მისი წვეროებისთვის დამთავრების დროს. საკმარისია ვაჩვენოთ, რომ თუ ნებისმიერი ორი განსხვავებული u და v წვეროსთვის არსებობს (u, v) წიბო, მაშინ $f[v] < f[u]$. (u, v) წიბოს დამუშავების მომენტში, წვერო v არ შეიძლება იყოს რუხი (ამ შემთხვევაში, ის იქნებოდა u-ს წინაპარი, ხოლო (u, v) წიბო იქნებოდა უკუწიბო, რაც ეწინააღმდეგება **ლემა** 2.5-ს, ამიტომ, (u, v) წიბოს დამუშავების მომენტში, წვერო v უნდა იყოს ან თეთრი, ან შავი. თუ v თეთრია, ის გახდება u-ს შთამომავალი და $f[v] < f[u]$. თუ ის უკვე შავია, მაშინ, $f[v]$-ს მნიშვნელობა უკვე დადგენილია, ხოლო u ჯერ დამუშავების პროცესშია, ამიტომ $f[v] < f[u]$. ამრიგად, აციკლური ორიენტირებული გრაფის ნებისმიერი (u, v) წიბოსთვის სრულდება $f[v] < f[u]$. □

2.5 ძლიერად ბმული კომპონენტები

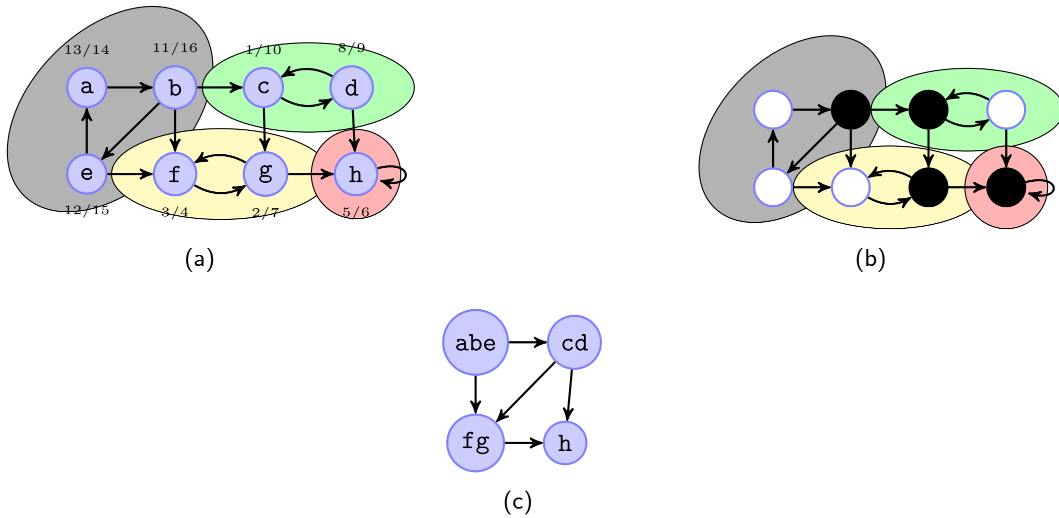
სიღრმეში ძებნის ალგორითმის გამოყენების კლასიკური მაგალითია გრაფის ძლიერად ბმულ კომპონენტებად დაშლა. ორიენტირებულ გრაფებზე მომუშავე მრავალი ალგორითმი იწყება გრაფში ძლიერად ბმული კომპონენტების მოძებნით. ამის შემდეგ ამოცანა იხსნება ცალკეული კომპონენტებისათვის, ხოლო შემდეგ ხდება კომბინირება ამ კომპონენტთა კავშირების შესაბამისად.
გავიხსენოთ, რომ $G = (V, E)$ ორიენტირებული გრაფის ძლიერად ბმული კომპონენტი ეწოდება $U \subset V$ წვეროთა მაქსიმალურ სიმრავლეს, სადაც ნებისმიერი ორი u და v წვერო $(u, v \in U)$ ერთმანეთისაგან მიღწევადია. $G = (V, E)$ გრაფის ძლიერად ბმული კომპონენტების ძებნის ალგორითმი იყენებს "ტრანსპონირებულ" $G^T = (V, E^T)$ გრაფს, რომელიც მიიღება საწყისი გრაფიდან წიბოთა მიმართულებების შებრუნებით. ასეთი გრაფი შეიძლება აიგოს $O(V + E)$ დროში (ვგულისხმობთ, რომ ორივე გრაფი მოიცემა მოსაზღვრე წვეროთა სიით). ცხადია, რომ G და G^T გრაფებს ერთი და იგივე ძლიერად ბმული კომპონენტები აქვთ. შემდეგი ალგორითმი პოულობს ძლიერად ბმულ კომპონენტებს $G = (V, E)$ ორიენტირებულ გრაფში სიღრმეში ძებნის ორჯერ გამოყენებით - G და G^T გრაფებისათვის. ალგორითმის მუშაობის დროა $O(V + E)$.

Algorithm 5: Strongly Connected Components

Input: ორიენტირებული გრაფი $G = (V, E)$
Output: ძლიერად ბმული კომპონენტების წვეროების სია

```
1 STRONGLY-CONNECTED-COMPONENTS(G) :
2     DFS(G);        // ვიპოვოთ f მასივი
3                    // წვეროების დამუშავების დამთავრების დროები
4     ავაგოთ G^T; // ავაგოთ გრაფის ტრანსპონირებული გრაფი
5     DFS(G^T);   // გარე ციკლში DFS str.6
6                 // წვეროები f კლებადობის მიხედვით ჩავიაროთ
7                 // აგებული ძებნის ხეები იქნება
8                 // G გრაფის ძლიერად ბმული კომპონენტები
9     return ძლიერად ბმული კომპონენტების სია
```

სურ. 2.8ა-ზე რუხი ფონით აღნიშნულია G გრაფის ძლიერად ბმული კომპონენტები, ნაჩვენებია აგრეთვე სიღრმეში ძებნის ტყე და დროის ჭდეები G^T გრაფისათვის. სურ. 2.8ბ-ზე მოცემულია G გრაფის სიღრმეში ძებნის ხე, რომელიც პროცედურის მე-5 სტრიქონში გამოითვლება. b, c, g, h წვეროები წარმოადგენენ სიღრმეში ძებნის ხეთა ფესვებს G^T გრაფისათვის და შეფერილი არიან შავად სურ. 2.8ც-ზე მოცემულია აციკლური ორიენტირებული გრაფი, რომელიც მიიღება G გრაფის ძლიერად ბმული კომპონენტების წერტილებამდე შეკუმშვით. მას **კომპონენტების გრაფს** უწოდებენ.
ამ ალგორითმის იდეა ეყრდნობა **კომპონენტების გრაფის** $G^{SCC} = (V^{SCC}, E^{SCC})$ თვისებას, რომელიც შემდეგში მდგომარეობს: ვთქვათ, G გრაფს აქვს ძლიერად ბმული კომპონენტები $C_1, \ldots, C_k$. წვეროთა $V^{SCC} = \{v_1, \ldots, v_k\}$ სიმრავლე შედგება v_i წვეროებისგან გრაფის ყოველი ძლიერად ბმული C_i კომპონენტისთვის. თუ G-ში არსებობს წიბო (x, y) რაიმე ორი წვეროსთვის $x \in C_i$, $y \in C_j$, მაშინ კომპონენტების გრაფში არსებობს წიბო $(v_i, v_j) \in E^{SCC}$. სხვა სიტყვებით რომ ვთქვათ, თუ G გრაფის ყოველ ძლიერად ბმულ კომპონენტში შევკუმშავთ მოსაზღვრე წვეროების დამაკავშირებელ წიბოებს, მივიღებთ G^{SCC} გრაფს, რომლის წვეროებს წარმოადგენენ G გრაფის ძლიერად ბმული კომპონენტები. იხ. სურ. 2.8ც.
კომპონენტების გრაფის ძირითადი თვისება მდგომარეობს იმაში, რომ ეს გრაფი წარმოადგენს აციკლურ ორიენტირებულ გრაფს, რაც გამომდინარეობს შემდეგი ლემიდან:



ნახ. 2.8:

ლემა 2.6. ვთქვათ, C და C' ორიენტირებული G გრაფის განსხვავებული ძლიერად ბმული კომპონენტებია, და ვთქვათ, $u, v \in C$ და $u', v' \in C'$, გარდა ამისა, ვთქვათ, G გრაფში არსებობს გზა u-დან u'-ში. მაშინ G გრაფში არ შეიძლება არსებობდეს გზა v'-დან v-ში.

Proof. თუ G გრაფში არსებობს გზა v'-დან v-ში, მაშინ G-ში არსებობს გზები u-დან v'-ში და v'-დან u-ში. ე.ი. u და v' ერთმანეთისთვის მიღწევადი წვეროებია, რაც ეწინააღმდეგება პირობას, რომ C და C' G გრაფის განსხვავებული ძლიერად ბმული კომპონენტებია. □

რადგან, პროცედურა STRONGLY-CONNECTED-COMPONENTS ორჯერ ასრულებს სიღრმეში ძებნას, ამ თავში ჩავთვალოთ, რომ $d[u]$ და $f[u]$ აღნიშნავენ აღმოჩენის და დამთავრების დროებს DFS პროცედურის პირველი გამოძახების შესრულების დროს (სტრ. 2).
შემოვიღოთ შემდეგი აღნიშვნები: ვთქვათ, $U \subseteq V$, $d(U) = \min_{u \in U}\{d[U]\}$, $f(U) = \max_{u \in U}\{f[u]\}$. ე.ი. $d(U)$ წარმოადგენს წვეროს აღმოჩენის ყველაზე ადრინდელ დროს U-ს წვეროებს შორის, ხოლო $f(U)$ - დამთავრების ყველაზე გვიან დროს U-ს წვეროებს შორის.

ლემა 2.7. ვთქვათ, C და C' ორიენტირებული $G = (V, E)$ გრაფის განსხვავებული ძლიერად ბმული კომპონენტებია, და ვთქვათ, არსებობს წიბო $(u, v) \in E$, სადაც $u \in C$ და $v \in C'$, მაშინ $f(C) > f(C')$.

Proof. იმის მიხედვით, სიღრმეში ძებნის პროცესში ძლიერად ბმულ რომელ კომპონენტში, C-ში თუ C'-შია პირველად აღმოჩენილი წვერო, გვაქვს ორი შემთხვევა:

- თუ $d(C) < d(C')$, აღვნიშნოთ C-ში აღმოჩენილი პირველი წვერო x-ით. $d[x]$ მომენტში ყველა წვერო C-ში და C'-ში თეთრია. G-ში არსებობს გზა x-დან C-ს ყველა წვერომდე, რომელიც შედგება მხოლოდ თეთრი წვეროებისგან. რადგან, $(u, v) \in E$, $d[x]$ მომენტში, ნებისმიერი $\omega \in C'$ წვეროსთვის, G-ში არსებობს აგრეთვე გზა x-დან ω-ში, რომელიც შედგება მხოლოდ თეთრი წვეროებისგან. თეთრი გზის შესახებ თეორემის თანახმად, ყველა წვერო C-ში და C'-ში, ხდება x-ს შთამომავალი სიღრმეში ძებნის ხეზე. **შედეგი** 2.2-ის თანახმად, $f[x] = f(C) > f(C')$.

- თუ $d(C) > d(C')$, აღვნიშნოთ C'-ში აღმოჩენილი პირველი წვერო y-ით. $d[y]$ მომენტში ყველა წვერო C'-ში თეთრია. G-ში არსებობს გზა y-დან C'-ს ყველა წვერომდე, რომელიც შედგება მხოლოდ თეთრი წვეროებისგან. თეთრი გზის შესახებ თეორემის თანახმად, ყველა წვერო C'-ში ხდება y-ს შთამომავალი სიღრმეში ძებნის ხეზე. **შედეგი** 2.2-ის თანახმად, $f[y] = f(C')$. $d[y]$ მომენტში ყველა წვერო C-ში თეთრია. რადგან არსებობს წიბო (u, v) C-დან C'-ში, **ლემა** 2.6-ის თანახმად, არ არსებობს გზა C'-დან C-ში. ე.ი. C-ში არ არის y-დან მიღწევადი წვეროები. ამგვარად, $f[y]$ მომენტში ყველა წვერო C-ში რჩება თეთრი, რაც ნიშნავს, რომ ნებისმიერი $\omega \in C$ წვეროსთვის, გვაქვს $f[\omega] > f[y]$, საიდანაც გამომდინარეობს, რომ $f[x] = f(C) > f(C')$.

□

შედეგი 2.3. ვთქვათ, C და C' ორიენტირებული $G = (V, E)$ გრაფის განსხვავებული ძლიერად ბმული კომპონენტებია, და ვთქვათ, არსებობს წიბო $(u, v) \in E^T$, სადაც $u \in C$ და $v \in C'$ მაშინ $f(C) < f(C')$.

Proof. რადგან $(u,v) \in E^T$ $(v,u) \in E$, G-ს და G^T-ს აქვს ერთი და იგივე ძლიერად ბმული კომპონენტები, **ლემა** 2.7-დან გამომდინარეობს, რომ $f(C) < f(C')$.
შედეგი 2.3-ს საშუალებით, განვიხილოთ როგორ მუშაობს პროცედურა STRONGLY-CONNECTED-COMPONENTS. როდესაც ვახორციელებთ სიღრმეში ძებნას G^T გრაფზე, ვიწყებთ იმ x წვეროდან, რომლისთვისაც $f[x]$ მაქსიმალურია. ეს წვერო ეკუთვნის რაიმე ძლიერად ბმულ C კომპონენტს. ამასთან, მოხდება C-ს ყველა წვეროს განხილვა. **შედეგი** 2.3-ის თანახმად, G^T გრაფში არ არის წიბო, რომელიც აკავშირებს C-ს სხვა ძლიერად ბმულ კომპონენტთან (რადგან მაშინ $f(C) < f(C')$, რაც ეწინააღმდეგება იმას, რომ $f[x]$ მაქსიმალურია.) ამიტომ x-დან სიღრმეში ძებნის დროს არ ხდება სხვა კომპონენტების წვეროების განხილვა. ამრიგად ხე, რომლის ფესვი არის x, შეიცავს მხოლოდ C-ს წვეროებს. მას შემდეგ, რაც განხილული იქნება C-ს ყველა წვერო, პროცედურის მე-5 სტრიქონში ხდება წვეროს არჩევა იმ სხვა ძლიერად ბმულ C' კომპონენტიდან, რომლისთვისაც $f(C')$ მაქსიმალურია ყველა სხვა კომპონენტთან შედარებით. შედეგი 2.3-ს თანახმად, G^T გრაფში ერთადერთი წიბო, რომელიც დააკავშირებდა C'-ს სხვა კომპონენტებთან შეიძლება იყოს მიმართული C-ში, მაგრამ C-ს დამუშავება უკვე დასრულებულია. ამრიგად, ყოველი სიღრმეში ძებნა ახორციელებს მხოლოდ ერთი ძლიერად ბმული კომპონენტის დამუშავებას. □

თეორემა 2.6. პროცედურა STRONGLY-CONNECTED-COMPONENTS(G) კორექტულად ახორციელებს ძლიერად ბმულ კომპონენტების ძებნას G გრაფში.

Proof. ვისარგებლოთ ინდუქციით სიღრმეში ძებნის დროს G^T გრაფში ხეების რაოდენობის მიხედვით და დავამტკიცოთ, რომ ყოველი ხის ფესვი ქმნის ძლიერად ბმულ კომპონენტს. $k = 0$-თვის ეს მტკიცება სამართლიანია.
დავუშვათ, სიღრმეში ძებნის პირველი k ხე (სტრ. 5) წარმოადგენს ძლიერად ბმულ კომპონენტს და განვიხილოთ $k+1$-ე ხე. ვთქვათ, ამ ხის ფესვია u და ვთქვათ, u ეკუთვნის C ძლიერად ბმულ კომპონენტს. რადგან G^T გრაფში სიღრმეში ძებნა ხორციელდება წვეროების $f[u]$ სიდიდის კლებადობის მიხედვით, ამიტომ ნებისმიერი C' ძლიერად ბმული კომპონენტისთვის, რომლის წვეროები ჯერ განხილული არ არის და რომელიც განსხვავებულია C-გან, სამართლიანია: $f[u] = f(C) > f(C')$. ინდუქციის დაშვების თანახმად, u წვეროს აღმოჩენის მომენტში, C-ს ყველა წვერო თეთრია, თეთრი გზის შესახებ თეორემის თანახმად, C-ს ყველა წვერო u-ს გარდა, წარმოადგენს u-ს შთამომავალს სიღრმეში ძებნის ხეზე. გარდა ამისა, G^T-ს ყველა წიბო, რომელიც გამოდის C-დან, მიმართული შეიძლება იყოს უკვე განხილული ძლიერად ბმული კომპონენტების წვეროებისკენ. ამიტომ, არც ერთ C-გან განსხვავებულ ძლიერად ბმულ კომპონენტში არ არის წვერო, რომელიც იქნებოდა u-ს შთამომავალი სიღრმეში ძებნის ხეზე. ამრიგად, G^T-ს u ფესვის მქონე სიღრმეში ძებნის ხის წვეროები, ქმნიან ზუსტად ერთ ძლიერად ბმულ კომპონენტს, რაც ამტკიცებს თეორემას. □

რიგი (Queue) არის დინამიკური სიმრავლე, რომელშიც ელემენტების ჩამატება და წაშლა ნებისმიერ პოზიციაში კი არ ხდება, არამედ განისაზღვრება სიმრავლის სტრუქტურით. რიგიდან შეიძლება მხოლოდ იმ ელემენტის წაშლა, რომელიც მასში პირველი იქნა ჩამატებული, ანუ რიგში ყველაზე დიდხანს იმყოფება, ხოლო ელემენტის ჩამატება ხდება რიგის ბოლოს: რიგი ორგანიზებულია პრინციპით: პირველი მოვიდა - ბოლო წავიდა ანუ FIFO (first-in, first-out).
Q რიგში v ელემენტის დამატების ოპერაცია აღინიშნება როგორც ENQUEUE(Q,v), ხოლო პირველი ელემენტის ამოშლა - DEQUEUE(Q), ამასთან ეს ოპერაცია აბრუნებს პირველი ელემენტის მნიშვნელობას. განისაზღვრება რიგის თავი (head) და ბოლო (tail). ყოველი ახალდამატებული ელემენტი აღმოჩნდება რიგის ბოლოში, ხოლო წასაშლელი თავში. head(Q) არის რიგის დასაწყისის ინდექსი, ხოლო $tail(Q)$ თავისუფალი უჯრის ინდექსი, რომელიც განკუთვნილია ახალი ელემენტის ჩასამატებლად. რიგი შედგება მასივის ელემენტებისგან, რომლებიც დგანან შესაბამისად $head(Q), head(Q)+1, \ldots, tail(Q)-1$ ადგილებზე.
სტეკი (stack) არის დინამიკური სიმრავლე, რომელშიც შეიძლება მხოლოდ ბოლო ელემენტის ამოშლა, ხოლო ელემენტის ჩამატება შეიძლება მხოლოდ მის ბოლოს. ორგანიზებულია პრინციპით: ბოლო მოვიდა - პირველი წავიდა ანუ LIFO (last-in, first-out).

2.6 სავარჯიშოები

1. სურ. 2.9 გრაფებისთვის განახორციელეთ სიგანეში ძებნა, ააგეთ სიგანეში ძებნის ხე.

2. სურ. 2.9 გრაფებისთვის განახორციელეთ სიღრმეში ძებნა. ააგეთ სიღრმეში ძებნის ტყე.

3. ვთქვათ, მოცემული გვაქვს $G = (V, E)$ არაორიენტირებული გრაფი. სამართლიანია თუ არა შემდეგი მტკიცებულებები:

   (ა) სიღრმეში ძებნის ყველა ტყე (დაწყებული სხვადასხვა საწყისი წვეროდან) შეიცავს ხეების ერთი და იგივე რაოდენობას.

(a) (b)

ნახ. 2.9:

(ზ) სიღრმეში ძებნის ყველა ტყე შეიცავს ხის წიბოების ერთი და იგივე რაოდენობას.

4. ვთქვათ, მოცემული გვაქვს $G = (V, E)$ ორიენტირებული გრაფი. აჩვენეთ, რომ (u, v) წიბო არის ხის წიბო ან პირდაპირი წიბო მაშინ და მხოლოდ მაშინ, როცა $d[u] < d[v] < f[v] < f[u]$.

5. სურ. 2.10ა გრაფში დაალაგეთ წვეროები ტოპოლოგიური სორტირების ალგორითმით.

(a) (b)

ნახ. 2.10:

6. $G = (V, E)$ გრაფში, სადაც $V = \{v, s, w, q, t, x, y, z\}$
$E = \{(v, w), (w, s), (s, v), (q, w), (q, s), (q, t), (t, y), (y, q), (t, x), (x, z), (z, x)\}$ განახორციელეთ სიღრმეში ძებნა და მოახდინეთ წიბოთა კლასიფიკაცია.

7. იპოვეთ ძლიერად ბმული კომპონენტები სურ. 2.10ბ გრაფში.

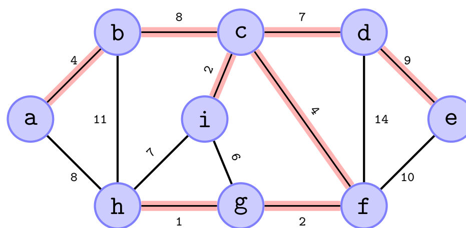
თავი 3

მინიმალური დამფარავი ხეები

ვთქვათ, მოცემულია ბმული არაორიენტირებული $G = (V, E)$ გრაფი. გრაფის ყოველი $(u, v) \in E$ წიბოსათვის მოცემულია $w(u, v)$ არაუარყოფითი წონა. ვიპოვოთ ისეთი ბმული, აციკლური ქვესიმრავლე $T \subseteq E$, რომელიც მოიცავს ყველა წვეროს და რომლისთვისაც ჯამური წონა

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

მინიმალურია. რადგან T სიმრავლე აციკლურია და აერთებს G გრაფის ყველა წვეროს, ის ქმნის ხეს, რომელსაც უწოდებენ ამ გრაფის **დამფარავ ხეს** (spanning tree). ასეთი T სიმრავლის პოვნის ამოცანას კი უწოდებენ ამოცანას **მინიმალური დამფარავი ხის** (minimum-spanning-tree problem) შესახებ. აქ სიტყვა "მინიმალური" აღნიშნავს "მინიმალურ შესაძლო წონას". შევნიშნოთ, რომ თუკი განვიხილავთ მხოლოდ ხეებს, მაშინ წონათა არაუარყოფითობის პირობა შეგვიძლია უგულებელვყოთ, რადგან ყველა დამფარავ ხეში წიბოთა ერთნაირი რაოდენობაა და შეგვიძლია ერთი და იგივე სიდიდით გავზარდოთ ყველა წიბოს წონა, რაც მათ დადებითად აქცევს.



ნახ. 3.1:

სურ. 3.1-ზე მოცემულია ბმული გრაფისა და მისი მინიმალური დამფარავი ხის მაგალითი, რომლის წიბოებიც გამოყოფილია. მინიმალური დამფარავი ხის ჯამური წონაა 37 და ასეთი ხე ერთადერთი არაა. თუკი (b, c) წიბოს შევცვლით (a, h) წიბოთი, მივიღებთ სხვა დამფარავ ხეს იმავე ჯამური წონით.
ჩვენ განვიხილავთ მინიმალური დამფარავი ხის პოვნის ორ ხერხს: პრიმისა და კრასკალის ალგორითმებს. ორივე ალგორითმი იყენებს "ხარბ" სტრატეგიას - ეძებს "ლოკალურად საუკეთესო" ვარიანტს მუშაობის ყოველ ბიჯზე.
ჩვენი ალგორითმების ზოგადი სქემა ასეთია: საძებნი დამფარავი ხე აიგება A სიმრავლეს ყოველ ბიჯზე თანდათანობით - თავდაპირველად ცარიელ ემატება თითო წიბო და შენარჩუნებულია თვისება (ამ თვისებას უწოდებენ ალგორითმის **ციკლის ინვარიანტს**) - "ციკლის ნებისმიერი იტერაციის წინ A სიმრავლე წარმოადგენს რომელიღაც მინიმალური დამფარავი ხის ქვესიმრავლეს". მორიგ ბიჯზე დამატებული (u, v) წიბო იმგვარად ამოირჩევა, რომ არ დაირღვეს ეს თვისება, ე.ი. $A \cup \{(u, v)\}$ ასევე უნდა იყოს მინიმალური დამფარავი ხის ქვესიმრავლე. ასეთ წიბოს უწოდებენ **უსაფრთხო წიბოს** (safe edge) A-სათვის.
ცხადია, რომ მთავარი პრობლემა მე-3 სტრიქონში უსაფრთხო წიბოს მოძებნაა, მაგრამ მანამდე შევნიშნოთ, რომ ასეთი წიბო გარანტირებულად არსებობს, რადგან მე-4 სტრიქონის შესრულების დროს, ციკლის ინვარიანტის თანახმად, უნდა არსებობდეს მინიმალური დამფარავი ხე T, რომ $A \subseteq T$. ამიტომ უნდა არსებობდეს წიბო $(u, v) \in T$, ისეთი, რომ $(u, v) \notin A$ და (u, v) უსაფრთხო წიბოა A ხისთვის.
ვიდრე უსაფრთხო წიბოს მოძებნის ალგორითმებს განვიხილავდეთ, განვსაზღვროთ რამდენიმე ტერმინი.
$G = (V, E)$ არაორიენტირებული გრაფის $(S, V \backslash S)$ **ჭრილი** (cut) ეწოდება მისი წვეროთა სიმრავლის გაყოფას ორ ქვესიმრავლედ.

Algorithm 6: Generic MST

Input: ბმული არაორიენტირებული გრაფი $G = (V, E)$ და წონის ფუნქცია $w : E \to R$
Output: გრაფის დამფარავი ხე

```
1 GENERIC-MST(G, w) :
2     A = ∅;
3     while ( A არ არის დამფარავი ხე ) :
4         მოვძებნოთ (u, v) უსაფრთხო წიბო A ხისთვის;
5         A = A ∪ {(u, v)};
6     return A
```

იტყვიან, რომ $(u, v) \in E$ წიბო **კვეთს** (crosses) $(S, V\backslash S)$ ჭრილს, თუ მისი ერთი ბოლო ეკუთვნის S-ს, ხოლო მეორე ბოლო - $(V\backslash S)$-ს. ჭრილი **შეთანხმებულია წიბოთა** A **სიმრავლესთან** (respects the set A), თუ A-დან არც ერთი წიბო არ კვეთს ამ ჭრილს. ჭრილის მიერ გადაკვეთილ წიბოთა სიმრავლეში გამოყოფენ უმცირესი წონის წიბოს, რომელსაც **მსუბუქს** (light edges) უწოდებენ.

თეორემა 3.1. ვთქვათ $G = (V, E)$ ბმული არაორიენტირებული გრაფია და მის წიბოთა სიმრავლეზე განსაზღვრულია ნამდვილი წონითი w ფუნქცია. ვთქვათ A წიბოთა სიმრავლეა, რომელიც წარმოადგენს G გრაფის რომელიმე მინიმალური დამფარავი ხის ქვესიმრავლეს. ვთქვათ $(S, V\backslash S)$ წარმოადგენს G გრაფის ისეთ ჭრილს, რომელიც შეთანხმებულია A-სთან, ხოლო (u, v) წიბო ამ ჭრილის მსუბუქი წიბოა. მაშინ (u, v) წიბო წარმოადგენს უსაფრთხო წიბოს A-სათვის.

Proof. ვთქვათ, T მინიმალური დამფარავი ხეა, რომელიც შეიცავს A-ს. დავუშვათ, T არ შეიცავს (u, v) წიბოს, რადგან წინააღმდეგ შემთხვევაში, დასამტკიცებელი დებულება ცხადია. ვაჩვენოთ, რომ არსებობს სხვა მინიმალური დამფარავი ხე T', რომელიც შეიცავს $A \cup \{(u, v)\}$. ამით დამტკიცდება, რომ (u, v) წიბო უსაფრთხოა A-სთვის.
T ბმულია, ამიტომ ის შეიცავს რაიმე p გზას (ერთადერთს) u-დან v-ში. (u, v) წიბო ქმნის ციკლს ამ გზასთან ერთად. რადგან u და v წვეროები ეკუთვნიან $(S, V\backslash S)$ ჭრილის სხვადასხვა ქვესიმრავლეს, p გზაზე არსებობს მინიმუმ ერთი წიბო, რომელიც კვეთს ჭრილს. ვთქვათ, ეს წიბოა (x, y). იგი არ ეკუთვნის A-ს, რადგან ჭრილი შეთანხმებულია A-სთან. დავუმატოთ T ხეს (u, v) წიბო და მიღებული ციკლიდან ამოვიღოთ (x, y) წიბო, მივიღებთ ახალ დამფარავ ხეს $T' = T\backslash\{(x, y)\} \cup \{(u, v)\}$.
ახლა, ვაჩვენოთ, რომ T' მინიმალური დამფარავი ხეა. რადგან (u, v) მსუბუქი წიბოა, რომელიც კვეთს $(S, V\backslash S)$ ჭრილს, T-დან ამოჭრილ (x, y) წიბოს, აქვს არანაკლები წონა, ვიდრე მის ნაცვლად დამატებულ (u, v)-ს. $(w(u, v) \leq w(x, y))$. ამიტომ T'-ს წონა შეიძლება მხოლოდ შემცირებულიყო,

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$$

მაგრამ T მინიმალური დამფარავი ხეა, ე.ი. T'-ც უნდა იყოს იგივე წონის სხვა მინიმალური დამფარავი ხე. ამიტომ T'-ში შემავალი (u, v) წიბო უსაფრთხოა. □

თეორემა 3.1-ის დახმარებით განვიხილოთ როგორ მუშაობს პროცედურა GENERIC-MST. ალგორითმის მუშაობის პროცესში, A სიმრავლე ყოველთვის აციკლურია (ის მინიმალური დამფარავი ხის ქვესიმრავლეა). ალგორითმის მუშაობის ყოველ მომენტში გრაფი $G_A = (V, A)$ წარმოადგენს ტყეს და მისი ნებისმიერი ბმული კომპონენტი არის ხე. (ზოგიერთი ხე შეიძლება შედგებოდეს მხოლოდ ერთი წვეროსგან, მაგ.: როცა ალგორითმი იწყებს მუშაობას, A სიმრავლე ცარიელია, ხოლო ტყე შეიცავს $|V|$ რაოდენობა ერთი წვეროს შემცველ ხეს). გარდა ამისა, A-ს ნებისმიერი უსაფრთხო (u, v) წიბო აკავშირებს G_A-ს სხვადასხვა კომპონენტს, რადგან სიმრავლე $A \cup \{(u, v)\}$ უნდა იყოს აციკლური.
პროცედურა GENERIC-MST-ში, ციკლი 3-5 სტრიქონებში სრულდება $|V| - 1$-ჯერ, რადგან ნაპოვნი უნდა იქნას მინიმალური დამფარავი ხის ყველა $|V| - 1$ წიბო. თავიდან, როცა $A = \varnothing$, G_A-ში არის ხეების $|V|$ რაოდენობა და ყოველი იტერაცია ამცირებს ამ რაოდენობას ერთით. როცა G_A-ში რჩება ერთი ხე, ალგორითმი სრულდება.

შედეგი 3.1. ვთქვათ $G = (V, E)$ ბმული არაორიენტირებული გრაფია და წიბოთა E სიმრავლეზე განსაზღვრულია ნამდვილი წონითი w ფუნქცია. ვთქვათ A წიბოთა სიმრავლეა, რომელიც წარმოადგენს G გრაფის რომელიმე მინიმალური დამფარავი ხის ქვესიმრავლეს. განვიხილოთ ტყე $G_A = (V, A)$ და ვთქვათ $C = (V_C, E_C)$ - მისი ერთ-ერთი ბმული კომპონენტია (ხეა). თუ (u, v) მსუბუქი წიბოა, რომელიც აკავშირებს C-ს G_A-ს რაიმე სხვა კომპონენტთან, მაშინ ეს წიბო უსაფრთხოა A-სთვის.

Proof. ჭრილი $(V_C, V\backslash V_C)$ შეთანხმებულია A-სთან და (u, v) მსუბუქი წიბოა ამ ჭრილისთვის, ამიტომ ის უსაფრთხოა A-სთვის. □

განვიხილოთ მინიმალური დამფარავი ხის პოვნის ორი ალგორითმი. თითოეული მათგანი იყენებს უსაფრთხო წიბოს ამორჩევის საკუთარ წესს (პროცედურა GENERIC-MST, სტრ. 4). **კრასკალის** ალგორითმში A სიმრავლე წარმოადგენს ტყეს, A-ს ემატება უსაფრთხო წიბოები, რომლებიც არიან ორი სხვადასხვა კომპონენტის დამაკავშირებელი მინიმალური წონის მქონე წიბოები. **პრიმის** ალგორითმში A სიმრავლე წარმოადგენს ერთიან ხეს, A-ს ემატება უსაფრთხო წიბოები, რომელთაც აქვთ მინიმალური წონა და რომლებიც აერთიანებენ ფესვის მქონე ხეს ამ ხის გარეთ მქონე წვეროებთან.

3.1 კრასკალის ალგორითმი

კრასკალის ალგორითმის მუშაობის ნებისმიერ მომენტში ამორჩეულ წიბოთა A სიმრავლე (დამფარავი ხის ნაწილი) არ შეიცავს ციკლებს. ალგორითმი ეძებს უსაფრთხო წიბოს ტყეში დასამატებლად, (u, v) მინიმალური წონის მქონე წიბოს პოვნით ყველა იმ წიბოს შორის, რომელიც აკავშირებს სხვადასხვა ხეს ტყეში. აღვნიშნოთ ორი ხე, რომელიც უკავშირდება ერთმანეთს (u, v) წიბოთი C_1-ით და C_2-ით. რადგან (u, v) წიბო მსუბუქია $(C_1, V \backslash C_1)$ ჭრილისთვის, **შედეგი** 3.2-დან გამომდინარეობს, რომ (u, v) წიბო უსაფრთხოა. სურ. 3.2-ზე ნაჩვენებია თუ როგორ მუშაობს ალგორითმი.
ალგორითმი MST-KRUSKAL(G,w) იყენებს სამ ოპერაციას, რომელიც მუშაობს თანაუკვეთ სიმრავლეებზე.
თანაუკვეთ სიმრავლეებზე განსაზღვრული მონაცემთა სტრუქტურა განისაზღვრება თანაუკვეთი დინამიკური სიმრავლეების $S = (S_1, S_2, \ldots, S_k)$ ნაკრებით. ყოველი სიმრავლის იდენტიფიცირება ხდება **წარმომადგენლით** (representative) რომელიც არის ამ სიმრავლის რაიმე ელემენტი. სიმრავლის ყოველი ელემენტი წარმოადგენს რაიმე ობიექტს. აღვნიშნოთ ეს ობიექტი x-ით. განვიხილოთ შემდეგი სამი ოპერაცია:

MAKE-SET(x) ("შევქმნათ სიმრავლე") - ქმნის ახალ სიმრავლეს, რომლის ერთადერთი ელემენტია x (ამიტომ x იქნება წარმომადგენელიც). რადგან სიმრავლეები არ უნდა თანაიკვეთონ, x ობიექტი არ უნდა შედიოდეს არც ერთ სხვა სიმრავლეში.

FIND-SET(x) ("მოვძებნოთ სიმრავლე") - პროცედურა აბრუნებს მიმთითებელს იმ სიმრავლის წარმომადგენელზე, რომელიც შეიცავს x ელემენტს.

UNION(x,y) - აერთიანებს x-ის და y-ის შემცველ დინამიკურ სიმრავლეებს (აღვნიშნოთ ისინი S_x-ით და S_y-ით) ახალ სიმრავლეში. იგულისხმება, რომ ოპერაციის შესრულებამდე აღნიშნული სიმარავლეები არ თანაიკვეთებოდნენ. მიღებული წარმომადგენელი არის $S_x \cup S_y$ სიმრავლის ელემენტი. ხოლო ძველი სიმრავლეები S_x და S_y წაიშლება.

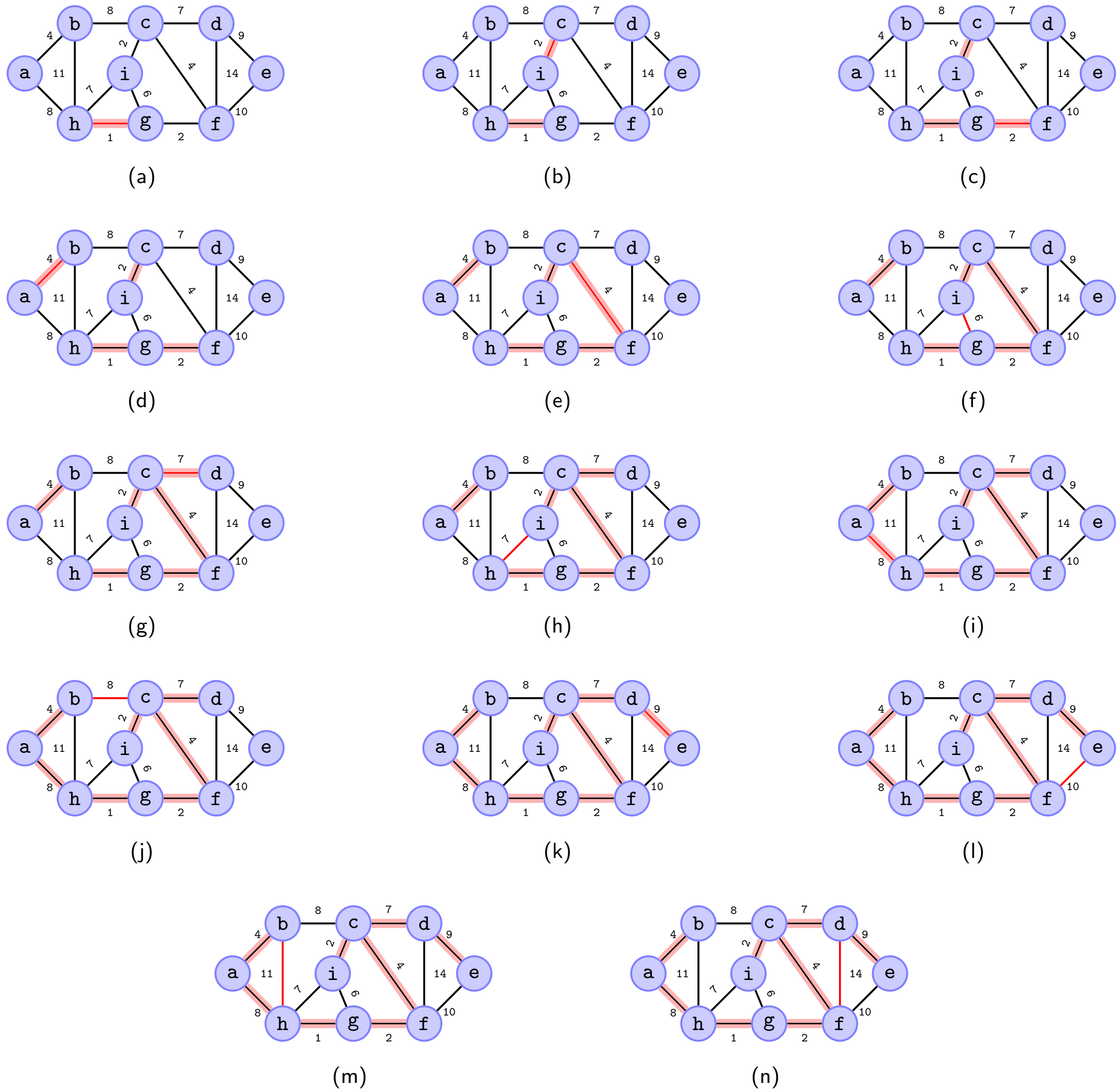
ზემოთ თქმულიდან გამომდინარე, გრაფის ორი u და v წვერო ეკუთვნის ერთ სიმრავლეს (ანუ კომპონენტს), როცა FIND-SET(u) = FIND-SET(v).

Algorithm 7: Minimum Spanning Tree - Kruskal

Input: ბმული არაორიენტირებული გრაფი $G = (V, E)$ და წონის ფუნქცია $w : E \to R$
Output: g რაფის მინიმალური წონის დამფარავი ხე

```
1  MST-KRUSKAL(G, w) :
2      A = ∅;
3      for ∀v ∈ V :
4          MAKE-SET(v);
5      sort(E); // დავალაგოთ წიბოების წონების ზრდის მიხედვით
6      for ∀(u,v) ∈ E :
7          if FIND-SET(u) ≠ FIND-SET(v) :
8              A = A ∪ {(u,v)};
9              UNION(u,v);
10     return A
```

ალგორითმის 2-4 სტრიქონებში ხდება A სიმრავლის ინიციალიზაცია ცარიელი სიმრავლით და იქმნება $|V|$ ხე, რომელიც შეიცავს თითო წვეროს. მე-5 სტრიქონში წიბოები ლაგდება წონათა არაკლებადობის მიხედვით. 6-9 სტრიქონებში ციკლი for ამოწმებს, ეკუთვნიან თუ არა წიბოს წვეროები ერთსა და იმავე ხეს. თუ ეკუთვნიან, მაშინ ამ წიბოს დამატება ტყისათვის არ შეიძლება (რადგან შეიქმნება ციკლი) და ხდება წიბოს უკუგდება, ხოლო თუ წიბოს წვეროები ეკუთვნიან სხვადასხვა ხეს, მაშინ წიბო ემატება A-ს (მე-8 სტრიქონი) და ამ წიბოთი დაკავშირებული ორი ხე ერთიანდება (მე-9 სტრიქონი).



ნახ. 3.2:

კრასკალის ალგორითმის მუშაობის დრო დამოკიდებულია თანაუკვეთ სიმრავლეებზე მონაცემთა სტრუქტურის რეალიზაციაზე. განვიხილოთ თანაუკვეთი სიმრავლეების წარმოდგენები ბმული სიების და ფესვის მქონე ხეების სახით.

თანაუკვეთი სიმრავლეები შეიძლება წარმოდგენილი იქნან ბმული სიების სახით. ყოველ ბმულ სიაში პირველი ობიექტი წარმოადგენს მის წარმომადგენელს. ამ ბმული სიის ყოველი ობიექტი შეიცავს სიმრავლის ელემენტს, მიმთითებელს სიმრავლის შემდეგი ელემენტის შემცველ ობიექტზე და მიმთითებელს წარმომადგენელზე. ყოველ ბმულ სიაში არის მიმთითებელი მის წარმომადგენელზე (head) და მიმთითებელი მის ბოლო ელემენტზე (tail).

თანაუკვეთი სიმრავლეების ასეთი წარმოდგენის შემთხვევაში, MAKE-SET(x) და FIND-SET(x) პროცედურებს სჭირდება $O(1)$ დრო. MAKE-SET(x) ქმნის ახალ ბმულ სიას ერთადერთი x ობიექტით, ხოლო FIND-SET(x) აბრუნებს მიმთითებელს x ელემენტის შემცველი სიმრავლის წარმომადგენელზე. UNION(x,y) პროცედურა სრულდება შემდეგნაირად: სია, რომელიც შეიცავს x ელემენტს, ემატება სიას, რომელიც შეიცავს y ელემენტს. y ელემენტის შემცველი სიის tail მიმთითებელი გამოიყენება იმისთვის, რომ სწრაფად განვსაზღვროთ სად დავამატოთ x-ის შემცველი სია. ახალი სიმრავლის მიმთითებელი ხდება y-ის შემცველი სიის მიმთითებელი. ამასთან უნდა განახლდეს მიმთითებლები x-ის შემცველი სიის ყოველი ობიექტისთვის, რაზეც დახარჯული დრო წრფივადაა დამოკიდებული x-ის შემცველი სიის სიგრძეზე.

ვთქვათ, გვაქვს $x_1, \ldots, x_n$ ობიექტი. სრულდება MAKE-SET(x)-ის n ოპერაცია და, შემდეგ, UNION(x,y)-ის $n-1$ ოპერაცია. MAKE-SET(x)-ის n ოპერაციას სჭირდება $\Theta(n)$ დრო, ხოლო, რადგან UNION(x,y)-ის i-ური ოპერაცია აახლებს i ობიექტს, UNION(x,y)-ის $n-1$ ოპერაციით განახლებული ობიექტების რაოდენობაა:

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

ოპერაციების მიმდევრობა ბმული სის შემთხვევაში	
ოპერაცია	განახლებული ობიექტების რაოდენობა
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
$\vdots$	$\vdots$
UNION(x_{n-1}, x_n)	$n-1$

ოპერაციების საერთო რაოდენობააა $2n-1$, ასე რომ საშუალოდ თითოეულ ოპერაციას სჭირდება $\Theta(n)$ დრო. UNION პროცედურას, წარმოდგენილი რეალიზაციით, უარეს შემთხვევაში, ერთი გამოძახებისთვის, საშუალოდ, სჭირდება $\Theta(n)$ დრო, რადგანაც შეიძლება აღმოჩნდეს, რომ გრძელ სიას ვაერთებთ მოკლე სიასთან და, ამიტომ უნდა განახლდეს ამ გრძელი სიის ყველა წევრის მიმთითებელი წარმომადგენელზე. დავუშვათ, ახლა, რომ ყოველი სია შეიცავს მისი სიგრძის აღმნიშვნელ ველს და ყოველთვის ვაერთებთ მოკლე სიას გრძელთან, მაშინ მტკიცდება, რომ MAKE-SET, FIND-SET, UNION m ოპერაციების მიმდევრობის შესრულებისთვის (რომელთაგანაც n მიმდევრობა MAKE-SET ოპერაციაა) საჭირო დრო ხდება $O(m + n \log n)$.

თუ თანაუკვეთ სიმრავლეებს წარმოვადგენთ ფესვის მქონე ხეების სახით, სადაც ყოველი კვანძი სიმრავლის ერთ ელემენტს შეესაბამება, ხოლო ყოველი ხე წარმოადგენს ერთ სიმრავლეს, მაშინ შესაძლებელია MAKE-SET, FIND-SET, UNION პროცედურების უფრო სწრაფი განხორციელება.

თანაუკვეთ სიმრავლეთა ტყეში (disjoint-set forest) ყოველი ელემენტი მიუთითებს მხოლოდ მშობელზე. ყოველი ხის ფესვი არის წარმომადგენელი და არის თავისი თავის მშობელიც.

თანაუკვეთ სიმრავლეებზე ოპერაციების ასიმტოტურად სწრაფად შესრულების საშუალებას იძლევა ორი ევრისტიკა: " **გაერთიანება რანგით**" და " **გზის შეკუმშვა**".

ოპერაციები შემდეგნაირად სრულდება:

MAKE-SET(x) - ქმნის ხეს ერთი კვანძით $\Theta(1)$ დროში, n რაოდენობა ერთ ელემენტიანი სიმრავლის შექმნას დასჭირდება $\Theta(n)$ დრო.

FIND-SET(x) - აბრუნებს x კვანძის შემცველი ხის x კვანძიდან ხის ფესვამდე გადაადგილებით (რომელიც არის წარმომადგენელი) მშობლების მიმთითებლის გავლით. თითოეულ ასეთ ოპერაციას სჭირდება $O(n)$ დრო. გავლილი კვანძები ამ გზაზე ქმნიან ძებნის გზას (find path).

UNION(x,y) - ხორციელდება y ხის ფესვის მიერთებით x ხის ფესვთან და y ხის ამოღებით ტყიდან. ამ ოპერაციას სჭირდება $\Theta(1)$ დრო.

გავეცნოთ ორ ევრისტიკას, რომელთა გამოყენებითაც შესაძლებელია დროითი საზღვრის შემცირება:

I **ევრისტიკა: გაერთიანება რანგით** (union by rank) დაფუძნებულია იდეაზე, რომ ყოველთვის, UNION ოპერაციის შესრულების დროს, ნაკლები რაოდენობის კვანძის შემცველი ხის ფესვი უერთდებოდეს მეტი რაოდენობის კვანძის შემცველი ხის ფესვს. ამისთვის გამოიყენება ფესვის რანგის (rank) ცნება. rank[x] არის x კვანძის სიმაღლე (x-დან მის შთამომავალ ფოთლამდე წიბოების რაოდენობა ყველაზე გრძელ გზაზე). ამ ევრისტიკის განსახორციელებლად, ყოველი კვანძისთვის შენახული უნდა გვქონდეს rank[x] (MAKE-SET პროცედურის მიერ ერთელემენტიანი სიმრავლის შექმნის დროს, rank[x]=0). FIND-SET არ ცვლის რანგებს. ადვილი დასამტკიცებელია, რომ რადგან ნებისმიერი კვანძის რანგი არ აღემატება $\lfloor \log n \rfloor$-ს, ძებნის ყოველი ოპერაცია ხორციელდება $O(\log n)$ დროში, ამრიგად, არა უმეტეს $n-1$ რაოდენობა UNION ოპერაციისა და m რაოდენობა FIND-SET ოპერაციის შესრულებას დასჭირდება $O(n + m \log n)$ დრო.

II **ევრისტიკა: გზის შეკუმშვა** (path compression) გამოიყენება FIND-SET ოპერაციის შესრულების პროცესში და ყველა კვანძს უშუალოდ უთითებს ფესვზე. ეს ევრისტიკა არ ცვლის კვანძების რანგებს.

ორი ხის გაერთიანების UNION პროცედურის გამოყენების დროს გვაქვს ორი შემთხვევა:

1. თუ ორ ხეს აქვს ერთნაირი რანგი, მაშინ ვირჩევთ ერთ-ერთი ხის ფესვს მშობლად და ვზრდით მის რანგს ერთით.
2. თუ ერთი ხის რანგი მეტია მეორე ხის რანგზე, მაშინ დიდი რანგის მქონე ხის ფესვი ხდება მშობელი კვანძი ნაკლები რანგის მქონე ხის ფესვისთვის, ხოლო რანგების მნიშვნელობები რჩება იგივე.

შემდეგ ფსევდოკოდებში $p[x]$ აღნიშნავს x-ს მშობელს.

Algorithm 8: Disjoint Set Data Structure Operations

```
1  MAKE-SET(x) :
2      p[x] = x;
3      rank[x] = 0;
4  FIND-SET(x) :
5      if x ≠ p[x] :
6          p[x] = FIND-SET(p[x]);
7      return p[x];
8  LINK(x,y) :
9      if rank[x] > rank[y] :
10         p[y] = x;
11     else:
12         p[x] = y;
13         if rank[x] == rank[y] :
14             rank[y] += 1;
15 UNION(x,y) :
16     LINK(FIND-SET(x), FIND-SET(y));
```

დავითვალოთ კრასკალის ალგორითმის მუშაობის დრო. ინიციალიზაციას სჭირდება დრო $O(V)$ (სტრ. 2-4). E წიბოების დალაგებას წონების მიხედვით - $O(E \log E)$ (სტრ. 5). 6-9 სტრიქონებში სრულდება $O(E)$ რაოდენობა FIND-SET და UNION ოპერაცია თანაუკვეთ სიმრავლეთა ტყეზე. V რაოდენობა MAKE-SET ოპერაციასთან ერთად, ამას სჭირდება $O((V+E)\alpha(V))$ დრო, სადაც α ძალიან ნელა ზრდადი ფუნქციაა. რადგან G ბმული გრაფია, სამართლიანია $|E| \geq |V|-1$ ასე, რომ, ოპერაციები თანაუკვეთ სიმრავლეებზე საჭიროებენ $O(E\alpha(V))$ დროს, გარდა ამისა, რადგან $\alpha(|V|) = O(\log V) = O(\log E)$, კრასკალის ალგორითმის მუშაობის დროა $O(E \log E)$. შევნიშნოთ, რომ $|E| < |V|^2$, ამიტომ $\log |E| = O(\log V)$ და კრასკალის ალგორითმის მუშაობის დრო შეიძლება ჩავწეროთ როგორც $O(E \log V)$.

3.2 პრიმის ალგორითმი

პრიმის ალგორითმი გამოირჩევა იმით, რომ A სიმრავლის წიბოები ყოველთვის ქმნიან ერთიან ხეს. პრიმის ალგორითმით მინიმალური დამფარავი ხის ფორმირება იწყება ნებისმიერი საწყისი r წვეროდან. ყოველ ბიჯზე ხეს

ემატება უმცირესი წონის წიბო იმ წიბოებს შორის, რომლებიც წვეროს აერთებენ ხის არაწევრ წვეროებთან. ზემოხსენებული შედეგის მიხედვით ასეთი წიბო უსაფრთხოა A-სათვის, ე.ი. მიიღება მინიმალური დამფარავი ხე.
პროცედურისათვის შემავალი მონაცემებია: ბმული G გრაფი, წიბოთა w წონები და r საწყისი წვერო. რეალიზაციისას მნიშვნელოვანია სწრაფად ავარჩიოთ მსუბუქი წიბო. ალგორითმის მუშაობისას ყველა წვერო, რომელიც ჯერ არ გამხდარა ხის წევრი, ინახება Q პრიორიტეტებიან რიგში. v წვეროს პრიორიტეტი განისაზღვრება $key[v]$ მნიშვნელობით, რომელიც უდრის იმ წიბოების მინიმალურ წონას, რომელიც აერთებს v-ს A ხესთან. თუ ასეთი წიბო არ არსებობს - $key[v] = \infty$. $\pi[v]$ ველი ხის წვეროებისთვის მიუთითებს მშობელს, ხოლო სხვა წვეროებისათვის ხის წვეროს, რომლისკენაც მივყავართ $key[v]$ წონის წიბოს (თუ ასეთი წიბო რამდენიმეა, მაშინ მიეთითება ერთ-ერთი).
ალგორითმის მუშაობის პროცესში, A სიმრავლე პროცედურა GENERIC-MST-ში არის შემდეგი:

$$A = \{(v, \pi[v]) : v \in V \backslash \{r\} \backslash Q\}$$

როცა ალგორითმი ასრულებს მუშაობას, Q პრიორიტეტებიანი რიგი ცარიელია, ხოლო მინიმალური დამფარავი ხე G-თვის არის ხე:

$$A = \{(v, \pi[v]) : v \in V \backslash \{r\}\}$$

Algorithm 9: Minimum Spanning Tree - Prim

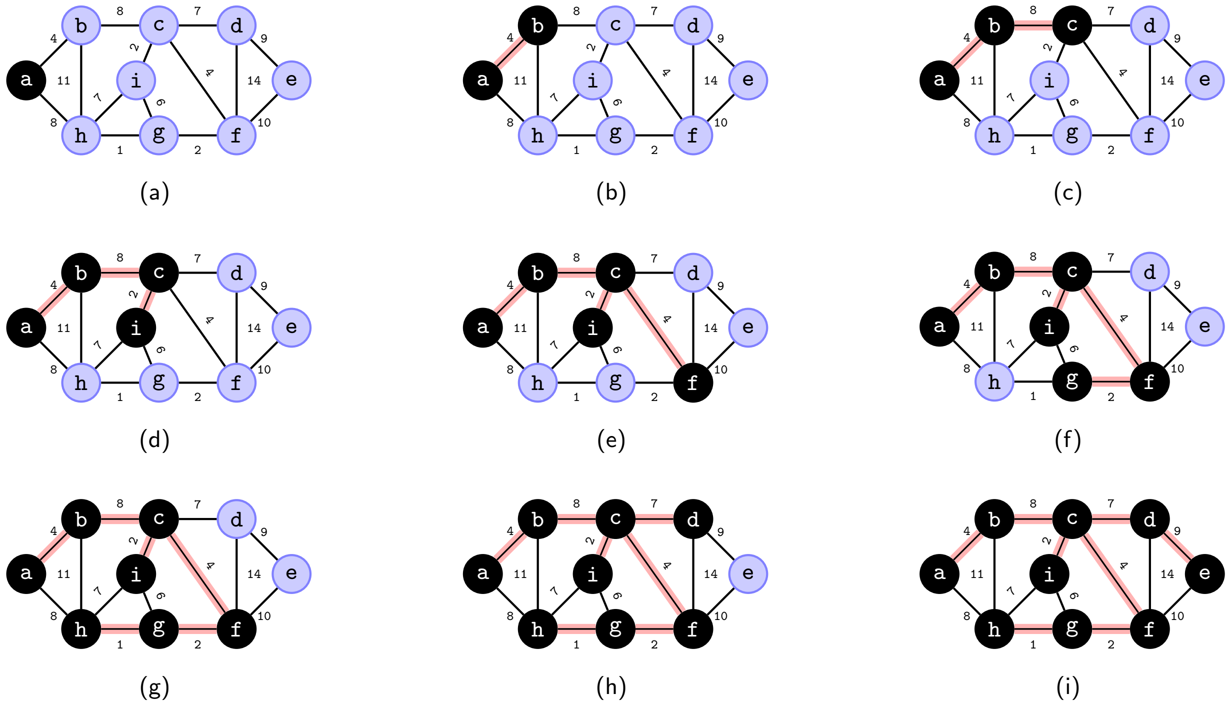
Input: ბმული არაორიენტირებული გრაფი $G = (V, E)$, წონის ფუნქცია $w : E \to R$ და საწყისი წვერო r
Output: გრაფის მინიმალური წონის დამფარავი ხე

```
1  MST-PRIM(G, w, r) :
2      for ∀u ∈ V :
3          key[u] = ∞;
4          π[u] = NIL;
5      key[r] = 0;
6      Q = V; // პრიორიტეტული რიგი, კეყ გვაძლევს პრიორიტეტს
7      while Q ≠ ∅ :
8          u = EXTRACT-MIN(Q);
9          for ∀v ∈ adj[u] :
10             if v ∈ Q and w(u,v) < key[v] :
11                 π[v] = u;
12                 key[v] = w(u,v);
13     return π; // ხე განისაზღვრება (v, π[v]) წიბოებით
```

2-6 სტრიქონებში ყველა წვეროს გასაღები ხდება ∞-ს ტოლი, გარდა r ფესვისა, რომლის გასაღები 0-ს ტოლია და ის აღმოჩნდება პირველი დასამუშავებელი წვერო. ყველა წვეროსთვის მშობლების ველში ჩაიწერება მნიშვნელობა NIL და ყველა წვერო ჩაიწერება Q პრიორიტეტებიან რიგში. while ციკლის ყოველი იტერაციის წინ, 7-12 სტრიქონებში:

- $A = \{(v, \pi[v]) : v \in V \backslash \{r\} \backslash Q\}$
- მინიმალურ დამფარავ ხეში უკვე შესული წვეროები ეკუთვნის $V \backslash Q$ სიმრავლეს
- ყველა $v \in Q$ წვეროსთვის, სამართლიანია: თუ $\pi[v] \neq NIL$, მაშინ $key[v] < \infty$ და $key[v]$ არის იმ $(v, \pi[v])$ მსუბუქი წიბოს წონა, რომელიც აკავშირებს v-ს რაიმე წვეროსთან, რომელიც უკვე მოთავსებულია მინიმალურ დამფარავ ხეში.

მე-8 სტრიქონში განისაზღვრება u წვერო, რომელიც ეკუთვნის $(V \backslash Q, Q)$ ჭრილის გადამკვეთ მსუბუქ წიბოს (პირველი იტერაციის გარდა). ხდება u-ს ამოღება Q-დან, და მისი დამატება ხის წვეროების $V \backslash Q$ სიმრავლეში. მოცემული ციკლის პირველი გავლისას ხე შედგება ერთადერთი წვეროსაგან. ყველა დანარჩენი წვერო იმყოფება რიგში. key[v]-ს მნიშვნელობა მათთვის r-დან v-ში წიბოს სიგრძის ან უსასრულობის (თუკი წიბო არ არსებობს) ტოლია. ალგორითმის მუშაობა მოცემულია სურ. 3.3-ზე. EXTRACT-MIN(Q) შლის მინიმალურ ელემენტს Q რიგიდან და აბრუნებს მას.
ალგორითმის მუშაობის დრო დამოკიდებულია Q პრიორიტეტებიანი რიგის რეალიზაციაზე. თუკი გამოყენებულია ორობითი გროვა, მაშინ მუშაობის დრო კრასკალის ალგორითმის ანალოგიურია - $O(E \log V)$. (2-6 სტრიქონებში ინიციალიზაცია შეიძლება შევასრულოთ $O(V)$ დროში. while ციკლი სრულდება $|V|$-ჯერ და ყოველი ოპერაცია EXTRACT-MIN სრულდება $O(\log V)$ დროში. EXTRACT-MIN-ის შესრულების საერთო დრო გახდება $O(V \log V)$. ციკლი 9-12 სტრიქონებში სრულდება $O(E)$-ჯერ. მე-9 სტრიქონში შემოწმება - $O(1)$ დროში. ხოლო მე-12 სტრიქონი



ნახ. 3.3:

შესრულდება $O(\log V)$ დროში. საბოლოოდ, პრიმის ალგორითმის მუშაობის საერთო დრო იქნება $O(V \log V + E \log V) = O(E \log V)$. ფიბონაჩის გროვის გამოყენების შემთხვევაში შეფასება შეიძლება შემცირდეს $O(E + V \log V)$-მდე.

3.3 სავარჯიშოები

1. იპოვეთ მინიმალური დამფარავი ხე სურ. 3.4 გრაფებისთვის:
   (ა) კრასკალის ალგორითმით
   (ბ) პრიმის ალგორითმით (c საწყისი წვეროდან)

ნახ. 3.4:

2. აჩვენეთ, რომ გრაფს აქვს ერთადერთი მინიმალური დამფარავი ხე, თუ გრაფის ყოველი ჭრილისთვის არსებობს ერთადერთი მსუბუქი წიბო, რომელიც კვეთს ამ ჭრილს. მოიყვანეთ კონტრმაგალითი, რომელიც აჩვენებს, რომ საწინააღმდეგო დებულება არ სრულდება.

3. ვთქვათ, (u, v) მინიმალური წონის მქონე წიბოა G გრაფში, აჩვენეთ, რომ (u, v) ეკუთვნის G გრაფის რომელიმე მინიმალურ დამფარავ ხეს.

4. $G(V, E)$ ბმული, არაორიენტირებული გრაფისთვის, განვიხილოთ წიბოების E სიმრავლე, რომლის ყოველი ელემენტი წარმოადგენს მსუბუქ წიბოს გრაფის რაიმე შესაძლო ჭრილისთვის. მოიყვანეთ მაგალითი, როცა ეს სიმრავლე არ ქმნის მინიმალურ დამფარავ ხეს.

$\alpha(n)$ ძალიან ნელა ზრდადი ფუნქციაა. რეალურ ამოცანებში, რომელშიც გამოიყენება თანაუკვეთი სიმრავლეები, $\alpha(n) \leq 4$.

თავი 4

უმოკლესი გზები ერთი წვეროდან

თუ მოცემულია შეწონილი გრაფი, ხშირად საჭიროა ხოლმე მის ორ წვეროს შორის უმოკლესი გზის დადგენა (ორ წვეროს შემაერთებელ ყველა შესაძლო გზას შორის ისეთის არჩევა, რომლის წიბოთა წონების ჯამი მინიმალურია).
ამ ამოცანის გადაჭრაზე ძალიან ბევრი სხვა ამოცანაა დამოკიდებული, მათ შორის:

- სატრანსპორტო ქსელებში ორ პუნქტს შორის უმოკლესი გზის პოვნა: თუ გრაფს განვიხილავთ როგორც ქალაქებს (წვეროები) და მათ შემაერთებელ გზებს (წიბოები), ან ქალაქში ქუჩებს (წიბოები) და მათ გადაკვეთებს (წვეროები), ერთი პუნქტიდან მეორეში გადასვლისათვის უმცირესი გზის გამოთვლა ამ ამოცანის გადაჭრით შეიძლება;

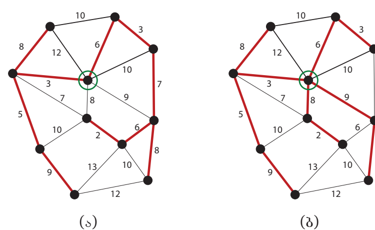
- ნალაპარაკევი ტექსტის ამოცნობის ერთ-ერთი უმთავრესი ამოცანა ერთნაირი ჟღერადობის სიტყვების (ომოფონების) განსხვავებაა. ასეთ სიტყვებზეა აგებული აკაკი წერეთელის ცნობილი ლექსი „აღმართ-აღმართ":

  აღმართ-აღმართ მივდიოდი ***მე ნელა***,
  სერზედ შევდექ, ჭმუნვის ალი ***მენელა***;
  მზემან სხივი მომაფინა ***მაშინა***,
  სიცოცხლე ვგრძენ, სიკვდილმა ***ვერ მაშინა***.

  თუ ენის სიტყვებს აღვნიშნავთ, როგორც გრაფის წვეროებს და „მსგავს" სიტყვებს წიბოებით შევაერთებთ (თანაც მსგავსების კოეფიციენტს წიბოს წონად მივუწერთ - რაც უფრო მსგავსია ორი სიტყვა, უფრო ნაკლებს), წვეროებს შორის უმოკლესი გზის პოვნა წინადადების აზრის დადგენაში დაგვეხმარება.

- გრაფთა განლაგებაში: ხშირად საჭიროა ხოლმე გრაფის „ცენტრის" დადგენა და ისე განლაგება, რომ იგი მის შუაგულში მოექცეს. ასეთი შეიძლება იყოს წვერო, რომლის მაქსიმალური დაშორება ყველა სხვა წვეროსთან ყველაზე დაბალია. ცხადია, რომ ამის დასადგენად საჭიროა ნებისმიერ ორ წვეროს შორის მანძილის ცოდნა.

აღსანიშნავია, რომ უმცირესი დამფარავი ხე ყოველთვის უმცირეს მანძილს არ მოგვცემს, როგორც ეს შემდგომ ნახაზშია ნაჩვენები.



ნახ. 4.1: მინიმალური დამფარავი ხე (ა) და შემოხაზული წვეროდან უმოკლესი მანძილის ხე (ბ)

<u>სავარჯიშო 4.1:</u> მოიყვანეთ სხვა ხეების მაგალითი, რომლებშიც უმცირესი დამფარავი ხე არ მოგვცემს უმოკლეს მანძილებს.

4.1 უმოკლესი გზის პოვნის ამოცანა

ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი ნამდვილი წონითი $w : E \to R$ ფუნქციით. $p =< v_0, v_1, \ldots, v_k >$ გზის **წონას** (weight) უწოდებენ ამ გზაში შემავალი ყველა წიბოს წონების ჯამს:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

u-დან v-ში **უმოკლესი გზის წონა** (shortest-paths weight), განსაზღვრების თანახმად, ტოლია:

$$\delta(u, v) = \begin{cases} \min\{w(p)\} & \text{თუ არსებობს გზა } u\text{-დან } v\text{-ში} \\ \infty & \text{წინააღმდეგ შემთხვევაში} \end{cases}$$

უმოკლესი გზა (shortest-path) u-დან v-ში - ესაა ნებისმიერი p გზა u-დან v-ში, რომლისთვისაც $w(p) = \delta(u, v)$. წონებში შეიძლება ვიგულისხმოთ არა მარტო მანძილები, არამედ დრო, ღირებულება, ჯარიმა, ზარალი და ა.შ. სიგანეში ძებნის ალგორითმი შეგვიძლია განვიხილოთ, როგორც უმოკლესი გზების შესახებ ამოცანის ამოხსნის კერძო შემთხვევა, როცა თითოეული წიბოს წონა 1-ის ტოლია.

განიხილავენ უმოკლესი გზების ამოცანის სხვადასხვა ვარიანტს. ამ თავში ჩვენ განვიხილავთ ამოცანას **უმოკლესი გზების შესახებ ერთი წვეროდან** (single-source shortest-path problem): მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი და საწყისი წვერო s (source vertex). საჭიროა ვიპოვოთ უმოკლესი გზები s-დან ყველა $v \in V$ წვერომდე. ამ ამოცანის ამოხსნის ალგორითმი გამოიყენება სხვა ამოცანების ამოსახსნელადაც, კერძოდ:

უმოკლესი გზა ერთი წვეროსაკენ: მოცემულია საბოლოო t წვერო (destination vertex). საჭიროა მოვძებნოთ უმოკლესი გზები t წვერომდე ყოველი $v \in V$ წვეროდან. თუკი შევაბრუნებთ მიმართულებას ყველა წიბოზე, ეს ამოცანა დაიყვანება ამოცანაზე უმოკლესი გზების შესახებ ერთი წვეროდან.

უმოკლესი გზა წვეროთა მოცემული წყვილისათვის: მოცემულია u და v წვეროები, მოვძებნოთ უმოკლესი გზა u-დან v-ში. რა თქმა უნდა, თუკი ჩვენ მოვძებნით ყველა უმოკლეს გზას u-დან, ამოცანა ამოიხსნება. უნდა აღინიშნოს, რომ უფრო სწრაფი მეთოდი (რომელიც გამოიყენებდა იმ ფაქტს, რომ უმოკლესი გზა მხოლოდ ორ წვეროს შორისაა მოსაძებნი) ჯერჯერობით ნაპოვნი არ არის.

უმოკლესი გზები წვეროთა ყველა წყვილისათვის: წვეროთა ყოველი u და v წყვილისათვის მოვძებნოთ უმოკლესი გზა u-დან v-ში. ამ ამოცანის ამოხსნა შეიძლება, თუკი რიგ-რიგობით ვიპოვით უმოკლეს გზას წვეროთა ყველა წყვილისათვის. თუმცა ეს არაა ოპტიმალური მეთოდი და უფრო ეფექტურ მიდგომას მომდევნო თავებში განვიხილავთ.

4.1.1 უმოკლესი გზების ამოცანის ოპტიმალური სტრუქტურა

უმოკლესი გზების პოვნის ალგორითმები, ჩვეულებრივ, ეყრდნობიან იმ თვისებას, რომ უმოკლესი გზის ყოველი ნაწილი თვითონ არის უმოკლესი გზა. ე.ი. უმოკლესი გზების ამოცანას აქვს ოპტიმალურობის თვისება ქვეამოცანებისათვის, რაც იმას ნიშნავს, რომ ამოცანის ამოსახსნელად შესაძლოა გამოყენებულ იქნას დინამიკური პროგრამირების მეთოდი ან ხარბი ალგორითმი. მართლაც, დეიქსტრას ალგორითმი ხარბ ალგორითმს წარმოადგენს, ხოლო ფლოიდ-ვორშელის ალგორითმი, რომელიც წვეროთა ყველა წყვილისთვის ეძებს უმოკლეს გზებს, დინამიკური პროგრამირების მეთოდს იყენებს.

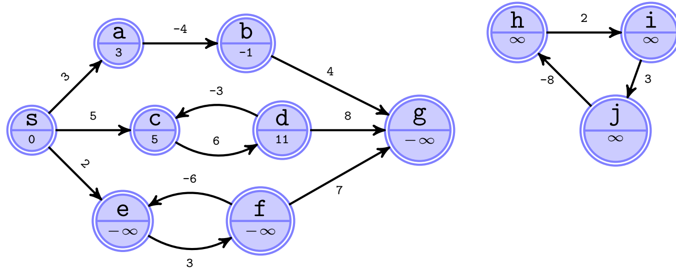
ლემა 4.1. (უმოკლესი გზის მონაკვეთები უმოკლესია). ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი წონითი $w : E \to R$ ფუნქციით. თუ $p =< v_1, v_2, \ldots, v_k >$ უმოკლესი გზაა v_1-დან v_k-მდე და $1 \leq i \leq j \leq k$, მაშინ $p_{ij} =< v_i, v_{i+1}, \ldots, v_j >$ წარმოადგენს უმოკლეს გზას v_i-დან v_j-მდე.

Proof. თუ p გზას დავშლით შემადგენელ ნაწილებად, $v_1 \overset{p_{1i}}{\sim} v_i \overset{p_{ij}}{\sim} v_j \overset{p_{jk}}{\sim} v_k$, შესრულდება შემდეგი: $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. ახლა დავუშვათ, რომ არსებობს გზა p'_{ij} v_i-დან v_j-მდე, რომლის წონაც აკმაყოფილებს უტოლობას: $w(p'_{ij}) < w(p_{ij})$. მაშინ $v_1 \overset{p_{1i}}{\sim} v_i \overset{p'_{ij}}{\sim} v_j \overset{p_{jk}}{\sim} v_k$ არის გზა v_1-დან v_k-მდე, რომლის წონა $w(p) = w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ ნაკლებია $w(p)$-ზე, რაც ეწინააღმდეგება პირობას, რომ p უმოკლესი გზაა v_i-დან v_j-მდე.

(ჩავთვალოთ, რომ ნებისმიერი ნამდვილი $a \neq -\infty$ რიცხვისთვის სრულდება: $a + \infty = \infty + a = \infty$ და ნებისმიერი ნამდვილი $a \neq \infty$ რიცხვისთვის სრულდება: $a + (-\infty) = (-\infty) + a = -\infty$) □

4.1.2 უარყოფითი წონის მქონე წიბოები

ზოგიერთ შემთხვევაში, წიბოთა წონები შესაძლოა უარყოფითი იყოს. ამ დროს დიდი მნიშვნელობა აქვს არსებობს თუ არა უარყოფითწონიანი ციკლი. თუ გრაფი არ შეიცავს s წვეროდან მიღწევად უარყოფითწონიან ციკლს, მაშინ ყოველი $v \in V$ წვეროსთვის $\delta(s, v)$ არის სასრული სიდიდე. ხოლო თუ s წვეროდან შესაძლებელია უარყოფითწონიან ციკლთან მისვლა, მაშინ არც ერთი გზა s წვეროდან ციკლის წვერომდე არ იქნება უმოკლესი, რადგან შეგვიძლია წონის განუწყვეტლივ შემცირება. ამრიგად, ასეთ შემთხვევაში უმოკლესი გზა არ არსებობს და თვლიან, რომ $\delta(s, v) = -\infty$.



ნახ. 4.2:

სურ. 4.2-ზე ნაჩვენებია რა გავლენას ახდენს უარყოფითწონიანი წიბოები და ციკლები უმოკლესი გზების წონებზე: რადგან, s წვეროდან a და b წვეროებამდე ერთადერთი გზა არსებობს, ამიტომ:

$$\delta(s, a) = w(s, a) = 3 \qquad \delta(s, b) = w(s, a) + w(a, b) = -1$$

s წვეროდან c წვერომდე უამრავი გზა არსებობს: $< s, c >, < s, c, d, c >, < s, c, d, c, d, c >$ და ა.შ., მაგრამ, რადგან $< c, d, c >$ ციკლის წონა დადებითია, უმოკლესი გზა s წვეროდან c წვერომდე არის $< s, c >$ და მისი წონაა $\delta(s, c) = 5$, ანალოგიურად, $\delta(s, d) = 11$.
s წვეროდან e წვერომდეც უამრავი გზა არსებობს: $< s, e >, < s, e, f, e >, < s, e, f, e, f, e >$ და ა.შ., მაგრამ, რადგან $< e, f, e >$ ციკლის წონა უარყოფითია, არ არსებობს უმოკლესი გზა s წვეროდან e წვერომდე და $\delta(s, e) = -\infty$, ანალოგიურად, $\delta(s, f) = -\infty$. რადგან g წვერო მიღწევადია f-დან, შეიძლება მოიძებნოს უსასრულოდ დიდი უარყოფითი წონის მქონე გზები s წვეროდან g წვერომდე, ამიტომ $\delta(s, g) = -\infty$.
h, i, j წვეროებიც ქმნიან უარყოფითწონიან ციკლს, მაგრამ ისინი არ არიან მიღწევადი s წვეროდან და ამიტომ $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

4.1.3 ციკლები

შეიძლება თუ არა უმოკლესი გზა შეიცავდეს ციკლს? დავრწმუნდით, რომ უმოკლესი გზა არ შეიძლება შეიცავდეს უარყოფითწონიან ციკლს. ის არ შეიძლება შეიცავდეს დადებითწონიან ციკლსაც, რადგან ამ ციკლის ამოღებით უმოკლესი გზიდან, მივიღებთ გზას საწყისი წვეროდან იმავე წვერომდე, რომელსაც აქვს ნაკლები წონა. თუ ციკლის წონა ნულია, მაშინ მისი ამოღებით უმოკლესი გზიდან, მივიღებთ გზას საწყისი წვეროდან იმავე წვერომდე, რომელსაც აქვს იგივე წონა და რომელიც არ შეიცავს ციკლს. ამიტომ, ზოგადობის შეუზღუდავად შეიძლება ჩავთვალოთ, რომ თუ ვეძებთ უმოკლეს გზებს, ისინი არ შეიცავენ ციკლებს. რადგან $G = (V, E)$ გრაფის ნებისმიერ აციკლურ გზაში შეიძლება შედიოდეს არაუმეტეს $|V|$ რაოდენობა წვეროებისა და $|V| - 1$ რაოდენობა წიბოებისა, შეიძლება შემოვიფარგლოთ უმოკლესი გზების პოვნით, რომელიც შეიცავს წიბოების არაუმეტეს $|V| - 1$ რაოდენობას.

4.1.4 უმოკლესი გზების წარმოდგენა

ზოგჯერ საჭირო ხდება არა მარტო უმოკლესი გზის წონის გამოთვლა, არამედ თავად ამ გზის დადგენაც. ასეთ შემთხვევაში იყენებენ იმავე მეთოდს, რომლითაც პოულობდნენ გზას სიგანეში ძებნის ხეებში. მოცემულ $G = (V, E)$ გრაფში, ყოველი $v \in V$ წვეროსთვის გამოითვლება $\pi[v]$ ატრიბუტის, მშობლის, **წინამორბედის** (predecessor) მნიშვნელობა, რომლის როლში გამოდის ან სხვა წვერო, ან მნიშვნელობა NIL. ამ თავში განხილული ალგორითმებით $\pi[v]$ ატრიბუტები ისე გამოითვლება, რომ v წვეროში დაწყებული წინამორბედების ჯაჭვი, გვაძლევს საშუალებას ავაგოთ s წვეროდან v წვეროში გამავალი უმოკლესი გზის შებრუნებული გზა. $G_\pi = (V_\pi, E_\pi)$

ქვეგრაფს უწოდებენ **წინამორბედობის ქვეგრაფს** (predecessor subgraph), სადაც

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\} \qquad E_\pi = \{(\pi[v], v) \in E : v \in V_\pi \backslash \{s\}\}$$

უმოკლესი გზის პოვნის ალგორითმების დასრულების შემდეგ, G_π წარმოადგენს "უმოკლესი გზების ხეს". ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი წონითი $w : E \to R$ ფუნქციით. ვთქვათ, გრაფი არ შეიცავს $s \in V$ საწყისი წვეროდან მიღწევად უარყოფითწონიან ციკლებს. s ფესვის მქონე **უმოკლესი გზების ხე** (shortest-paths tree) არის $G' = (V', E')$ ორიენტირებული ქვეგრაფი, სადაც $V' \subseteq V$, $E' \subseteq E$ განისაზღვრება შემდეგი პირობებით:

1. V' - არის წვეროთა სიმრავლე, რომელიც მიღწევადია $s \in V$ საწყისი წვეროდან G გრაფში.
2. G' გრაფი წარმოადგენს ხეს s წვეროს მქონე ფესვით.
3. ყოველი $v \in V'$ წვეროსთვის ცალსახად განსაზღვრული მარტივი გზა s წვეროდან v წვეროში G' გრაფში, ემთხვევა უმოკლეს გზას s წვეროდან v წვეროში G გრაფში.

4.1.5 რელაქსაცია

რელაქსაციის მექანიზმი ასეთია: თითოეული $v \in V$-სათვის ვინახავთ რაღაც $d[v]$ რიცხვს, ატრიბუტს, რომელიც წარმოადგენს s-დან v-ში უმოკლესი გზის წონის ზედა შეფასებას, ანუ, უბრალოდ უმოკლესი გზის შეფასებას (shortest-path estimate). d და π მასივებს საწყისი მნიშვნელობები ენიჭებათ შემდეგი პროცედურით, რომლის მუშაობის დროა $\Theta(V)$:

Algorithm 10: Initialize Single Source

Input: ორიენტირებული გრაფი $G = (V, E)$ და საწყისი წვერო s
Output: მიანიჭებს საწყისს მნიშვნელობებს d და π მასივებს

```
1 INITIALIZE-SINGLE-SOURCE(G, s) :
2     for ∀v ∈ V :
3         d[v] = ∞;
4         π[v] = NIL;
5     d[s] = 0;
```

$(u, v) \in E$ წიბოს რელაქსაცია შემდეგში მდგომარეობს: მოწმდება შეიძლება თუ არა v წვერომდე აქამდე არსებული უმოკლესი გზის გაუმჯობესება მისი u წვეროზე გატარებით? დადებითი პასუხის შემთხვევაში ხდება $d[v]$ და $\pi[v]$ ატრიბუტების განახლება. რელაქსაცია ამცირებს $d[v]$-ს $d[u] + w(u, v)$-მდე. ამავდროულად იცვლება $\pi[v]$-ც.

Algorithm 11: Relaxation

Input: ორიენტირებული გრაფი $G = (V, E)$, წონითი ფუნქცია $w : E \to R$, გრაფის წვეროები u და v
Output: ითვლის შემოწმების მომენტში უკეთესია თუ არა, რომ v წვეროში მოვხვდეთ u წვეროდან (u, v) წიბოს გამოყენებით

```
1 RELAX(u, v, w) :
2     if d[v] > d[u] + w(u,v) :
3         d[v] = d[u] + w(u,v);
4         π[v] = u;
```

ამ თავში აღწერილი ალგორითმებში ჯერ ხდება ინიციალიზაცია და შემდეგ რელაქსაცია. რელაქსაცია ერთადერთი პროცედურაა, რომელიც ცვლის $d[v]$ და $\pi[v]$ ატრიბუტებს. თუმცა ალგორითმები განსხვავდებიან იმით, თუ რამდენჯერ ტარდება რელაქსაცია და წიბოთა რა თანმიმდევრობისთვის. მაგ.: დეიქსტრას ალგორითმი აციკლური გრაფებისათვის მხოლოდ ერთხელ ახდენს წიბოთა რელაქსაციას, ხოლო ბელმან-ფორდის ალგორითმი - რამდენჯერმე.
განვიხილოთ უმოკლესი გზების და რელაქსაციას თვისებები, რომლებიც მოცემულია შემდეგ დებულებებში:

ლემა 4.2. (სამკუთხედის უტოლობა (triangle property)) ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი წონითი $w : E \to R$ ფუნქციით და $s \in V$ საწყისი წვეროთი. მაშინ ნებისმიერი $(u, v) \in E$-სათვის, გვაქვს:

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

Proof. ვთქვათ, არსებობს უმოკლესი p გზა s-დან v წვეროში, მაშინ ამ გზის წონა არ აღემატება ნებისმიერი სხვა გზის წონას s-დან v-ში, კერძოდ, ის არ აღემატება გზის წონას, რომელიც შედგება გზისგან s-დან u წვეროში და (u,v) წიბოსგან.
თუ უმოკლესი გზა s-დან v წვეროში არ არსებობს, მაშინ $\delta(s,v)=\infty$ ან $\delta(s,v)=-\infty$. $\delta(s,v)=\infty$ ნიშნავს, რომ წვერო არ არის მიღწევადი s-დან, მაგრამ მაშინ u წვეროც არ იქნება მიღწევადი $\delta(s,u)=\infty$ და დასამტკიცებელი უტოლობა სრულდება. თუ $\delta(s,v)=-\infty$, ეს ნიშნავს, რომ v წვერო არის უარყოფითწონიანი ციკლის წვერო და დასამტკიცებელი უტოლობა სრულდება. □

ლემა 4.3. (ზედა საზღვრის თვისება (upper-bound property)**)** ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G=(V,E)$ გრაფი წონითი $w:E\to R$ ფუნქციით. ვთქვათ, $s\in V$ - საწყისი წვეროა. მაშინ INITIALIZE-SINGLE-SOURCE(G,s) პროცედურის შესრულებისა და წიბოთა ნებისმიერი თანმიმდევრობით რელაქსაციის შემდეგ, ნებისმიერი $v\in V$ წვეროსათვის სრულდება უტოლობა $d[v]\geq\delta(s,v)$. თუკი რომელიმე წვეროსათვის ეს უტოლობა გადაიქცევა ტოლობად, $d[v]=\delta(s,v)$ მაშინ იგი აღარ შეიცვლება.

Proof. დავამტკიცოთ $d[v]\geq\delta(s,v)$, ყოველი $v\in V$ წვეროსთვის ინდუქციის მეთოდის გამოყენებით, რელაქსაციის ბიჯების რაოდენობის მიმართ. $d[v]\geq\delta(s,v)$ უტოლობა სამართლიანია უშუალოდ ინიციალიზაციის შემდეგ, რადგან საწყისი წვეროსთვის, $d[s]=0\geq\delta(s,s)$ (შევნიშნოთ, რომ $\delta(s,s)=-\infty$, თუ s წვერო უარყოფითწონიანი ციკლის წვეროა; წინააღმდეგ შემთხვევაში, $\delta(s,s)=0$), ხოლო $v\in V\backslash\{s\}$ წვეროებისთვის, ინიციალიზაციის შემდეგ, $d[v]=\infty$ და ამიტომ $d[v]\geq\delta(s,v)$.
ინდუქციის ბიჯად ჩავთვალოთ (u,v) წიბოს რელაქსაცია. ინდუქციის დაშვების თანახმად, ყველა $x\in V$ წვეროსთვის, რელაქსაციის წინ სრულდება უტოლობა $d[x]\geq\delta(s,x)$. დავამტკიცოთ, რომ უტოლობა სრულდება რელაქსაციის შემდეგაც. რელაქსაციის შემდეგ შეიძლება შეიცვალოს მხოლოდ $d[v]$, თუ ის შეიცვალა გვექნება:

$$d[v]=d[u]+w(u,v)\geq\delta(s,u)+w(u,v)\geq\delta(s,v)$$

სადაც პირველი უტოლობა ინდუქციის დაშვებიდან გამომდინარეობს, ხოლო მეორე - სამკუთხედის უტოლობიდან. ამრიგად, დამტკიცდა, რომ ნებისმიერი წვეროსათვის სრულდება უტოლობა $d[v]\geq\delta(s,v)$.
დავამტკიცოთ, რომ $d[v]$-ს მნიშვნელობა არ შეიცვლება მას შემდეგ, რაც შესრულდება $d[v]=\delta(s,v)$. რადგან $d[v]\geq\delta(s,v)$ სამართლიანია, $d[v]$ ვერ მიიღებს $\delta(s,v)$-ზე ნაკლებ მნიშვნელობას. $d[v]$-ს მნიშვნელობა ვერც გაიზრდება, რადგან რელაქსაციის შედეგად ის შეიძლება მხოლოდ შემცირდეს ან დარჩეს იგივე. □

ლემა 4.4. (არარსებული გზის თვისება (no-path property)**)**: ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G=(V,E)$ გრაფი, წონითი $w:E\to R$ ფუნქციით და s საწყისი წვეროთი. ვთქვათ, $v\in V$ წვერო მიუღწევადია s-დან. მაშინ INITIALIZE-SINGLE-SOURCE(G,s) პროცედურის შესრულების შემდეგ გვაქვს $d[v]=\delta(s,v)=\infty$ და ეს ტოლობა არ შეიცვლება G გრაფში წიბოთა ნებისმიერი თანმიმდევრობით რელაქსაციის შემდეგაც.

Proof. ზედა საზღვრის თვისების თანახმად, სრულდება $\infty=\delta(s,v)\leq d[v]$, ამიტომ $d[v]=\infty=\delta(s,v)$. □

ლემა 4.5. ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G=(V,E)$ გრაფი წონითი $w:E\to R$ ფუნქციით და ვთქვათ $(u,v)\in E$. მაშინ უშუალოდ ამ წიბოს რელაქსაციის შემდეგ სრულდება უტოლობა $d[v]\leq d[u]+w(u,v)$.

Proof. თუ უშუალოდ (u,v) წიბოს რელაქსაციამდე სრულდება $d[v]>d[u]+w(u,v)$, მაშინ ამ ოპერაციის შემდეგ, უტოლობა გადაიქცევა ტოლობად: $d[v]=d[u]+w(u,v)$, ხოლო თუ (u,v) წიბოს რელაქსაციამდე სრულდება $d[v]\leq d[u]+w(u,v)$, მაშინ რელაქსაციის პროცესში, არც $d[u]$, არც $d[v]$ არ შეიცვლება. ასე, რომ (u,v) წიბოს რელაქსაციის შემდეგ, $d[v]\leq d[u]+w(u,v)$. □

ლემა 4.6. (კრებადობის თვისება (convergence property)**)** ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G=(V,E)$ გრაფი, წონითი $w:E\to R$ ფუნქციით და s საწყისი წვეროთი. ვთქვათ, $u,v\in V$ წვეროებისთვის არსებობს უმოკლესი გზა $s\sim u\to v$. ვთქვათ, შესრულდა INITIALIZE-SINGLE-SOURCE(G,s) პროცედურა, ხოლო შემდეგ წიბოთა გარკვეული თანმიმდევრობით რელაქსაცია, მათ შორის RELAX(u,v,w). თუ რაღაც მომენტში (u,v) წიბოს რელაქსაციამდე შესრულდა $d[u]=\delta(s,u)$ ტოლობა, მაშინ (u,v) წიბოს რელაქსაციის შემდეგ, ნებისმიერ მომენტში, შესრულდება ტოლობა $d[v]=\delta(s,v)$.

Proof. ზედა საზღვრის თვისების თანახმად, თუ რაღაც მომენტში (u,v) წიბოს რელაქსაციამდე შესრულდა $d[u]=\delta(s,u)$ ტოლობა, ის შემდგომაც არ შეიცვლება. კერძოდ, (u,v) წიბოს რელაქსაციის შემდეგ. მივიღებთ:

$$d[v]\leq d[u]+w(u,v)=\delta(s,u)+w(u,v)=\delta(s,v)$$

სადაც უტოლობა გამომდინარეობს **ლემა** 4.5-დან, ხოლო ბოლო ტოლობა **ლემა** 4.1-დან. ზედა საზღვრის თვისების თანახმად, $d[v]\geq\delta(s,v)$. ამიტომ, შეიძლება დავასკვნათ, რომ $d[v]=\delta(s,v)$ და ის აღარ შეიცვლება. □

ლემა 4.7. (გზის რელაქსაციის თვისება (path-relaxation property)) ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი წონითი $w : E \to R$ ფუნქციით და s საწყისი წვეროთი. განვიხილოთ ნებისმიერი უმოკლესი გზა $p =< v_0, v_1, \dots, v_k > s = v_0$ წვეროდან v_k წვერომდე. ვთქვათ, შესრულდა INITIALIZE-SINGLE-SOURCE(G,s) პროცედურა, ხოლო შემდეგ, წიბოების შემდეგი თანმიმდევრობით რელაქსაცია:
$(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, მაშინ ამ რელაქსაციების შემდეგ და, შემდგომაც, ნებისმიერ მომენტში, სრულდება ტოლობა $d[v_k]\delta(s, v_k)$. ეს თვისება სამართლიანია, მიუხედავად იმისა, ხდება თუ არა რელაქსაცია სხვა წიბოებზე, მათ შორის რელაქსაცია იმ წიბოებზე, რომლებიც ენაცვლება p გზის წიბოებს.

Proof. ინდუქციით დავამტკიცოთ, რომ p გზის i-ური წიბოს რელაქსაციის შემდეგ, სრულდება: $d[v_i] = \delta(s, v_i)$. ბაზისად ავიღოთ $i = 0$. მანამ, სანამ p გზაში შემავალი ერთი მაინც წიბოს რელაქსაცია მოხდება, ინიციალიზაციის შემდეგ, $d[v_0] = d[s] = 0 = \delta(s, s)$. ზედა საზღვრის თვისების თანახმად, $d[s]$-ს მნიშვნელობა აღარ შეიცვლება.
ვთქვათ, $(i - 1)$-ური წიბოს რელაქსაციის შემდეგ სრულდება: $d[v_{i-1}] = \delta(s, v_{i-1})$. განვიხილოთ i-ური, (v_{i-1}, v_i) წიბოს რელაქსაცია. კრებადობის თვისების თანახმად, ამ წიბოს რელაქსაციის შედეგად, $d[v_i] = \delta(s, v_i)$ და ეს ტოლობა აღარ შეიცვლება. □

ლემა 4.8. (წინამორბედობის ქვეგრაფის თვისება (predecessor subgraph property)) ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი, წონითი $w : E \to R$ ფუნქციით და s საწყისი წვეროთი. ვთქვათ, G გრაფი არ შეიცავს s წვეროდან მიღწევად უარყოფითწონიან ციკლებს. ვთქვათ, შესრულდა INITIALIZE-SINGLE-SOURCE(G,s) პროცედურა, ხოლო შემდეგ, რაიმე თანმიმდევრობით G გრაფის წიბოების რელაქსაცია, რომლის შედეგადაც ყოველი $v \in V$ წვეროსთვის სრულდება ტოლობა $d[v] = \delta(s, v)$, მაშინ წინამორბედობის ქვეგრაფი G_π წარმოადგენს s ფესვის მქონე უმოკლესი გზების ხეს.

4.2 ბელმან-ფორდის ალგორითმი

ბელმან-ფორდის ალგორითმი (Bellman-Ford algorithm) ხსნის საწყისი წვეროდან უმოკლესი გზების პოვნის ამოცანას ზოგად შემთხვევაში, როცა ნებისმიერ წიბოს შესაძლოა ჰქონდეს უარყოფითი წონა. ამ ალგორითმის ღირსებად შეიძლება ჩაითვალოს ისიც, რომ ის განსაზღვრავს არსებობს თუ არა გრაფში საწყისი წვეროდან მიღწევადი უარყოფითწონიანი ციკლი. ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი წონითი $w : E \to R$ ფუნქციით და s საწყისი წვეროთი. ბელმან-ფორდის ალგორითმი იძლევა TRUE მნიშვნელობას, თუ გრაფში საწყისი წვეროდან არაა მიღწევადი უარყოფითწონიანი ციკლი და იძლევა მნიშვნელობას FALSE, თუკი ასეთი ციკლი საწყისი წვეროდან მიღწევადია. პირველ შემთხვევაში ალგორითმი პოულობს უმოკლეს გზებს და მათ წონებს, ხოლო მეორე შემთხვევაში - უმოკლესი გზა არ არსებობს.

Algorithm 12: Bellman-Ford (Single Source Shortest Paths)

Input: ორიენტირებული გრაფი $G = (V, E)$, წონითი ფუნქცია $w : E \to R$ და საწყისი წვერო s
Output: d-ში გამოითვლის უმოკლეს მანძილებს s-დან ყველა სხვა წვერომდე, π-ში გამოითვლის ხეს, ალგორითმი დააბრუნებს TRUE-ს თუ გრაფში არ არსებობს უარყოფითი წონის ციკლი, წინააღმდეგ შემთხვევაში FALSE-ს

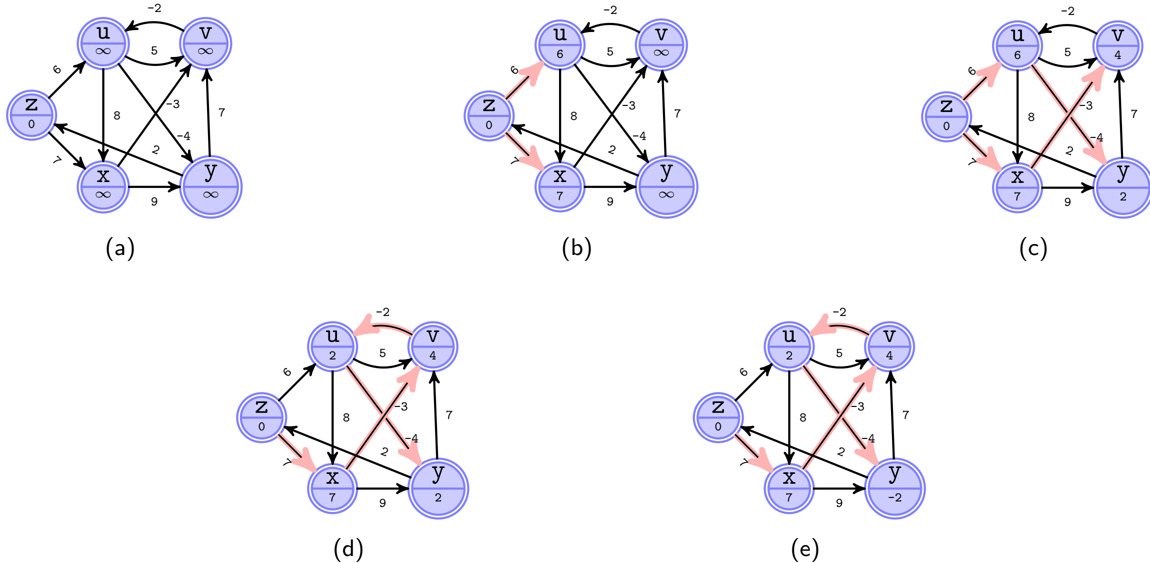
```
1 BELLMAN-FORD(G, w, s) :
2     INITIALIZE-SINGLE-SOURCE(G, s);
3     for i=1; i<|V|; i++ :
4         for ∀(u,v) ∈ E :
5             RELAX(u, v, w);
6     for ∀(u,v) ∈ E :
7         if d[v] > d[u] + w(u,v) :
8             return FALSE;
9     return TRUE;
```

ვნახოთ, როგორ მუშაობს ალგორითმი. სტრ. 2-ში ხდება ინიციალიზაცია. შემდეგ ალგორითმი $(|V| - 1)$-ჯერ იმეორებს ერთსა და იმავე მოქმედებას: ახდენს გრაფის თითოეული წიბოს რელაქსაციას (3-5 სტრიქონები). შემდეგ, ალგორითმი ამოწმებს, არსებობს თუ არა საწყისი წვეროდან მიღწევადი უარყოფითწონიანი ციკლი (6-8 სტრიქონები) და აბრუნებს შესაბამის მნიშვნელობას.
სურ. 4.3-ზე მოცემულია ბელმან-ფორდის ალგორითმის მუშაობის პროცესი. საწყისი წვეროა z. წვეროებში ნაჩვენებია უმოკლესი გზების შეფასებები (ატრიბუტი d), ხოლო გამოყოფილი წიბოები მიუთითებენ მშობლების

მნიშვნელობებს: თუ გამოყოფილია (u, v) წიბო, $\pi(v) = u$. ამასთან, წვეროების დამუშავება ხდება ლექსიკოგრაფიულად დალაგებული შემდეგი თანმიმდევრობით:

$$(u, v)(u, x)(u, y)(v, u)(x, v)(x, y)(y, v)(y, z)(z, u)(z, x)$$

ა)-ზე ნაჩვენებია ინიციალიზაციის პროცედურის შემდეგ, უშუალოდ წიბოების რელაქსაციის წინ არსებული სიტუაცია, ბ)-ც)-დ)-ზე ნაჩვენებია წიბოების რელაქსაციის შედეგად მიღებული მდგომარეობა, ხოლო ე)-ზე - საბოლოო მდგომარეობა.



ნახ. 4.3:

ბელმან-ფორდის ალგორითმის მუშაობის დროა $O(VE)$. ინიციალიზაციას სჭირდება $\Theta(V)$ დრო; 3-5 სტრიქონებში, თითოეულ ჯერზე წიბოების რელაქსაციას - $\Theta(E)$ დრო (სულ $(|V|-1)$-ჯერ); 6-8 სტრიქონებში ციკლის შესრულებას კი - $O(E)$.
ქვემოთ მოყვანილი თეორემა და მისი შედეგი, ამტკიცებს ბელმან-ფორდის ალგორითმის კორექტულობას.

ლემა 4.9. ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი წონითი $w : E \to R$ ფუნქციით და s საწყისი წვეროთი, რომელიც არ შეიცავს s წვეროდან მიღწევად უარყოფითწონიან ციკლს, მაშინ ბელმან-ფორდის ალგორითმის 3-5 სტრიქონებში, for ციკლის $(|V| - 1)$ იტერაციის დასრულების შემდეგ, ყოველი $v \in V$ s-დან მიღწევადი წვეროსთვის სრულდება $d[v] = \delta(s, v)$.

Proof. განვიხილოთ რაიმე s-დან მიღწევადი წვერო $v \in V$. ვთქვათ, $p =< v_0, v_1, \ldots, v_k >$, $v_0 = s$ $v_k = v$ უმოკლესი აციკლური გზაა s-დან v-ში. p გზა შეიცავს არაუმეტეს $(|V|-1)$ წიბოს, რაც ნიშნავს, რომ $k \le |V|-1$. for ციკლის (3-5 სტრ.) ყოველი $(|V|-1)$ იტერაციის დროს, ხდება ყველა $|E|$ წიბოს რელაქსაცია, i-ური $(i = 1, 2, \ldots, k)$ იტერაციის დროს რელაქსირებულ წიბოებს შორის არის წიბო (v_{i-1}, v_i). ამიტომ, **ლემა** 4.7-ს თანახმად, სრულდება: $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. □

შედეგი 4.1. ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი წონითი $w : E \to R$ ფუნქციით და s საწყისი წვეროთი. მაშინ, ყოველი $v \in V$ წვეროსთვის, გზა s-დან v-ში არსებობს მაშინ და მხოლოდ მაშინ, როცა G გრაფის ბელმან-ფორდის ალგორითმით დამუშავების შემდეგ, სრულდება: $d[v] < \infty$.

თეორემა 4.1. (ბელმან-ფორდის ალგორითმის კორექტულობა) ვთქვათ, ბელმან-ფორდის ალგორითმით ხდება s საწყისი წვეროს და $w : E \to R$ წონითი ფუნქციის მქონე ორიენტირებული, წონადი $G = (V, E)$ გრაფის დამუშავება. თუ G გრაფი არ შეიცავს s წვეროდან მიღწევად უარყოფითწონიან ციკლებს, მაშინ ალგორითმი აბრუნებს TRUE მნიშვნელობას, ყოველი $v \in V$ წვეროსთვის, სრულდება $d[v] = \delta(s, v)$ და წინამორბედობის ქვეგრაფი G_π არის s ფესვის მქონე უმოკლესი გზების ხე. ხოლო თუ G გრაფი შეიცავს s წვეროდან მიღწევად უარყოფითწონიან ციკლს, მაშინ ალგორითმი აბრუნებს FALSE მნიშვნელობას.

Proof. დავუშვათ, G გრაფი არ შეიცავს s წვეროდან მიღწევად უარყოფითწონიან ციკლს. ჯერ დავამტკიცოთ, რომ ალგორითმის მუშაობის დასრულების შემდეგ, ყოველი $v \in V$ წვეროსთვის, სრულდება $d[v] = \delta(s, v)$. თუ v წვერო მიღწევადია s-დან, ეს ტოლობა გამომდინარეობს **ლემა** 4.9-დან, ხოლო თუ v არ არის მიღწევადი s-დან - არარსებული გზის თვისებიდან. ამ დებულებიდან და წინამორბედობის ქვეგრაფის თვისებიდან (**ლემა** 4.8)

გამომდინარეობს, რომ G_π გრაფი არის უმოკლესი გზების ხე. ახლა, ვაჩვენოთ, რომ ბელმან-ფორდის ალგორითმი აბრუნებს TRUE მნიშვნელობას. ალგორითმის დასრულების შემდეგ, ყოველი $(u, v) \in E$ წიბოსთვის სრულდება:

$$d[v] = \ delta(s, v) \leq \delta(s, u) + w(u, v) = d[u] + w(u, v)$$

ამიტომ სტრ. 7-ში, არც ერთი შედარების შედეგად ალგორითმი არ დააბრუნებს FALSE მნიშვნელობას.
ახლა, განვიხილოთ შემთხვევა, როცა G გრაფი შეიცავს s წვეროდან მიღწევად უარყოფითწონიან ციკლს. ვთქვათ, ეს ციკლია $c =< v_0, v_1, \ldots, v_k >$, სადაც $v_0 = v_k$, მაშინ:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \tag{4.1}$$

დავუშვათ საწინააღმდეგო, ვთქვათ, ალგორითმი აბრუნებს TRUE მნიშვნელობას. (რაც, აგრეთვე, ნიშნავს, რომ ალგორითმი აბრუნებს უმოკლეს გზებს და მათ წონებს) მაშინ ყოველი $(i = 1, 2, \ldots, k)$-სთვის სრულდება: $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$. განვიხილოთ ამ უტოლობის ჯამი ციკლის ყველა წვეროსთვის:

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

რადგანაც, $v_0 = v_k$, ამიტომ c ციკლის ყოველი წვერო $\sum_{i=1}^{k} d[v_i]$ და $\sum_{i=1}^{k} d[v_{i-1}]$ ჯამებში გვხვდება მხოლოდ ერთხელ და $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$. **შედეგი** 4.10-ის თანახმად $d[v_i]$ ატრიბუტი იღებს სასრულ მნიშვნელობებს, ამრიგად სამართლიანია უტოლობა $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$, რაც ეწინააღმდეგება (4.1)-ს. მივიღეთ წინააღმდეგობა. ამრიგად, ბელმან-ფორდის ალგორითმი აბრუნებს TRUE მნიშვნელობას, თუ გრაფი არ შეიცავს საწყისი წვეროდან მიღწევად უარყოფითწონიან ციკლს და აბრუნებს FALSE მნიშვნელობას, წინააღმდეგ შემთხვევაში. □

4.3 უმოკლესი გზები აციკლურ ორიენტირებულ გრაფში

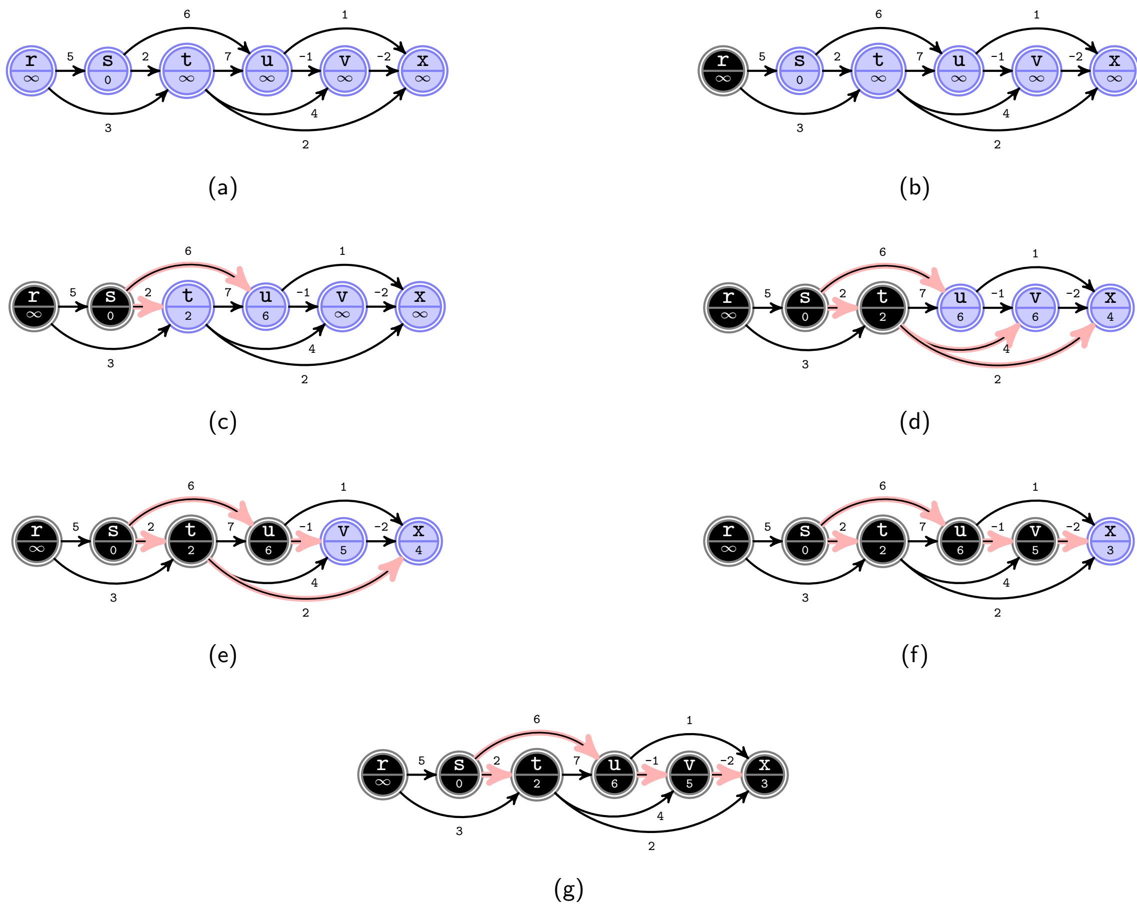
აციკლურ ორიენტირებულ $G = (V, E)$ გრაფში ერთი წვეროდან უმოკლესი გზების მოსაძებნად საჭიროა $\Theta(V + E)$ დრო, თუ წიბოების რელაქსაციას ჩავატარებთ ტოპოლოგიურად სორტირებული წვეროების მიხედვით. შევნიშნოთ, რომ აციკლურ ორიენტირებულ გრაფში უმოკლესი გზები ყოველთვის განსაზღვრულია, რადგან ციკლები (მათ შორის, უარყოფითწონიანიც) საერთოდ არ გვხვდება.
ტოპოლოგიური სორტირება იმგვარად განალაგებს წვეროებს წრფივი მიმდევრობით, რომ ყველა წიბო ერთნაირი მიმართულებისაა. ამის შემდეგ უნდა განვიხილოთ წვეროები ამ მიმდევრობით (წიბოს დასაწყისი ყოველთვის მის ბოლოზე ადრე იქნება განხილული) და ყოველი წვეროსათვის მოვახდინოთ მისგან გამომავალი ყველა წიბოს რელაქსაცია.

Algorithm 13: DAG Shortest Paths (Single Source Shortest Paths)

Input: აციკლური ორიენტირებული გრაფი $G = (V, E)$, წონითი ფუნქცია $w : E \to R$ და საწყისი წვერო s
Output: d-ში გამოითვლის უმოკლეს მანძილებს s-დან ყველა სხვა წვერომდე, π-ში გამოითვლის ხეს

```
1 DAG-SHORTEST-PATHS(G, w, s) :
2     L = TOPOLOGICAL-SORT(G);
3     INITIALIZE-SINGLE-SOURCE(G, s);
4     for ∀u ∈ L :
5         for ∀v ∈ Adj[u] :
6             RELAX(u, v, w);
7     return d, π;
```

ალგორითმის მუშაობის პროცესი ნაჩვენებია სურ. 4.4-ზე. საწყისი წვეროა s. ტოპოლოგიურ სორტირებაზე (სტრ. 2) იხარჯება $\Theta(V + E)$ დრო, ხოლო ინიციალიზაციაზე (მე-3 სტრ.) - $\Theta(V)$. წვეროსთვის ციკლში (4-6 სტრ.) სრულდება ერთი ოპერაცია, ყოველი წვეროდან გამომავალი წიბოები ერთხელ დამუშავდება, ამრიგად, შიდა ციკლში სულ სრულდება $|E|$ იტერაცია (5-6 სტრ.) თითოეული იტერაციის ღირებულება $\Theta(1)$-ია. ალგორითმის მუშაობის დროა $\Theta(V + E)$ და ის გამოისახება მოსაზღვრე წვეროთა სიის ზომის წრფივი ფუნქციით.



ნახ. 4.4:

აღწერილი ალგორითმი შესაძლებელია გამოვიყენოთ ე.წ. "კრიტიკული გზების" საპოვნელად. განვიხილოთ ამოცანა, სადაც ორიენტირებული, აციკლური გრაფის ყოველი წიბო წარმოადგენს რაღაც საქმიანობას, ხოლო წიბოს წონა - მის შესასრულებლად საჭირო დროს. თუკი გვაქვს წიბოები (u,v) და (v,x), მაშინ (u,v) წიბოს შესაბამისი სამუშაო უნდა შესრულდეს (v,x) წიბოს შესაბამისი სამუშაოს დაწყებამდე. **კრიტიკული გზა** (critical path) - ესაა უგრძელესი გზა გრაფში, რომლის წონა ტოლია ყველა სამუშაოს შესასრულებლად დახარჯული დროსი, თუკი მაქსიმალურადაა გამოყენებული ზოგიერთი სამუშაოს პარალელურად შესრულების შესაძლებლობა. კრიტიკული გზის წონა არის ყველა სამუშაოს შესრულების სრული დროის ქვედა შეფასება. კრიტიკული გზის საპოვნელად ყველა წონის ნიშანი უნდა შეიცვალოს საპირისპიროთი და შესრულდეს DAG-SHORTEST-PATHS ალგორითმი.

4.4 დეიქსტრას ალგორითმი

დეიქსტრას ალგორითმი პოულობს $G=(V,E)$ ორიენტირებული გრაფისათვის უმოკლეს გზებს საწყისი s წვეროდან ყველა დანარჩენ წვერომდე. აუცილებელია, რომ ყველა წიბოს წონა იყოს არაუარყოფითი $w(u,v)\geq 0$ ყოველი $(u,v)\in E$.

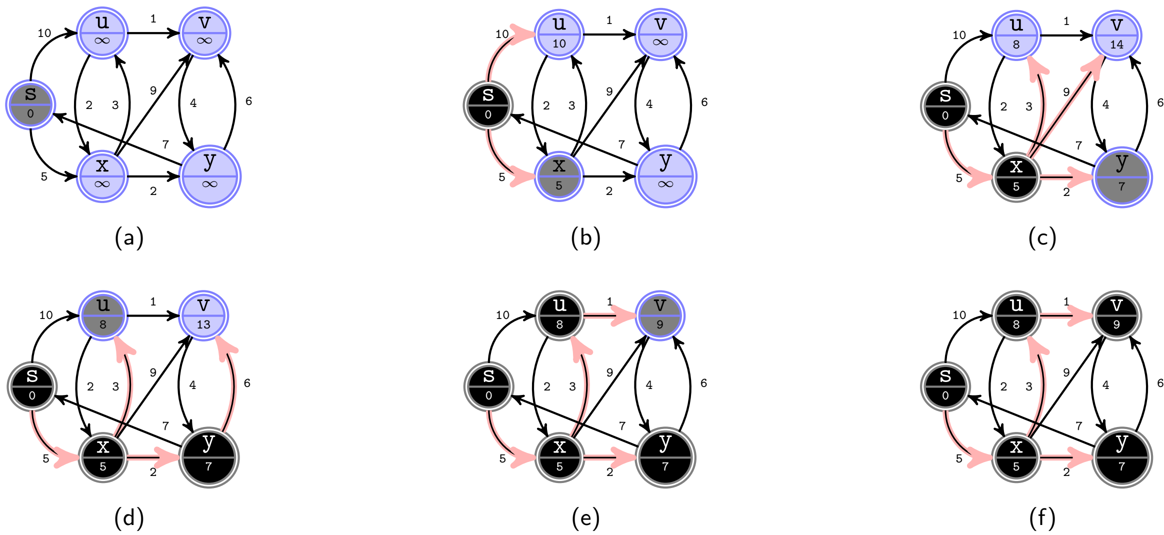
დეიქსტრას ალგორითმის მუშაობის დროს გამოიყენება $S\subseteq V$ სიმრავლე, რომელიც შედგება იმ v წვეროებისაგან, რომელთათვისაც $\delta(s,v)$ უკვე მოძებნილია (ე.ი. $d[v]=\delta(s,v)$). ალგორითმი ირჩევს უმცირესი $d[u]$-ს მქონე $u\in V\backslash S$ წვეროს, ამატებს u-ს S სიმრავლეში და ახდენს u-დან გამომავალი ყველა წიბოს რელაქსაციას, რის შემდეგაც ციკლი მეორდება. წვეროები, რომლებიც S-ს არ მიეკუთვნებიან, ინახება Q პრიორიტეტებიან რიგში, რომლის გასაღებიც განისაზღვრება d ფუნქციის მნიშვნელობებით. იგულისხმება, რომ გრაფი მოცემულია მოსაზღვრე

წვეროთა სიით.

Algorithm 14: Dijkstra Shortest Paths

Input: აციკლური ორიენტირებული გრაფი $G = (V, E)$, წონითი ფუნქცია $w : E \to R^+$ და საწყისი წვერო s
Output: d-ში გამოითვლის უმოკლეს მანძილებს s-დან ყველა სხვა წვერომდე, π-ში გამოითვლის ხეს

```
1  DIJKSTRA(G, w, s) :
2      INITIALIZE-SINGLE-SOURCE(G, s);
3      S = ∅;
4      Q = V; // პრიორიტეტული რიგი, დ გვაძლევს პრიორიტეტს
5      while Q ≠ ∅ :
6          u = EXTRACT-MIN(Q);
7          S = S ∪ {u};
8          for ∀v ∈ adj[u] :
9              RELAX(u, v, w);
10     return d, π;
```



ნახ. 4.5:

დეიქსტრას ალგორითმის მუშაობის პროცესი აღწერილია სურ. 4.5-ზე. საწყისი წერტილია s. წვეროებში ჩაწერილია უმოკლესი გზების შეფასებები მოცემული მომენტისათვის. შავი ფერით აღნიშნულია წვეროები, რომლებიც S სიმრავლეს ეკუთვნიან. სხვა წვეროები დგანან $Q = V \setminus S$ პრიორიტეტებიან რიგში. რუხი ფერის წვეროები ციკლის მომდევნო იტერაციის დროს გამოდიან u წვეროს როლში. d და π თავიანთ საბოლოო მნიშვნელობებს დებულობენ სურ. 4.5ფ-ზე.
ალგორითმის მუშაობის სტრ. 2-ში ხდება d-ს და π-ს, მე-3 სტრიქონში - S-ის, ხოლო მე-4 სტრიქონში - Q-ს ინიციალიზაცია. while ციკლის ყოველი იტერაციის წინ, 5-9 სტრ-ში $Q = V \setminus S$. დასაწყისში $Q = V$. 5-9 სტრიქონებში while ციკლის ყოველი იტერაციის დროს Q-დან ხდება უმცირესი $d[u]$-ს მქონე u წვეროს ამოღება და ის ემატება S სიმრავლეს (თავდაპირველად $u = s$). 8-9 სტრიქონებში ხდება u-დან გამოსული ყოველი (u, v) წიბოს რელაქსაცია. ამ დროს შეიძლება შეიცვალოს $d[v]$ შეფასება და $\pi[v]$ მშობელი. შევნიშნოთ, რომ ციკლის მუშაობის დროს Q რიგში ახალი წვეროები არ ემატება, ხოლო Q-დან ამოღებული ყოველი წვერო ემატება S სიმრავლეს მხოლოდ ერთხელ, ამიტომ while ციკლის იტერაციათა რაოდენობაა $|V|$.
რადგან დეიქსტრას ალგორითმში S სიმრავლეს ემატება $V \setminus S$ სიმრავლიდან ამოღებული ყველაზე "მსუბუქი" წვერო, ამიტომ ამბობენ, რომ ალგორითმი მუშაობს ხარბი სტრატეგიით, რომელიც ყოველთვის არ იძლევა ოპტიმალურ შედეგს. ქვემომოყვანილი თეორემა და მისი შედეგი, რომელიც დაუმტკიცებლად მოგვყავს, ამტკიცებს დეიქსტრას ალგორითმის კორექტულობას.

თეორემა 4.2. (დეიქსტრას ალგორითმის კორექტულობა) s საწყისი წვეროს და წონითი $w : E \to R$ ფუნქციის მქონე ორიენტირებული, წონადი $G = (V, E)$ გრაფის, დეიქსტრას ალგორითმით დამუშავების შემდეგ, ყოველი $u \in V$-სათვის სრულდება $d[u] = \delta(s, u)$.

შედეგი 4.2. s საწყისი წვეროს და წონითი $w : E \to R$ ფუნქციის მქონე ორიენტირებული, წონადი $G = (V, E)$

გრაფის დეიქსტრას ალგორითმით დამუშავების შემდეგ, წინამორბედობის ქვეგრაფი G_π წარმოადგენს უმოკლესი გზების ხეს, რომლის ფესვიც არის s წვერო.

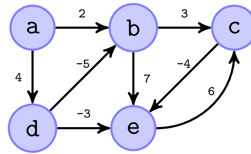
დეიქსტრას ალგორითმის მუშაობის დრო. ამ ალგორითმში გამოიყენება პრიორიტეტებიანი Q რიგი და სამი ოპერაცია (INSERT (სტრ. 4), EXTRACT-MIN (სტრ. 6), DECREASE-KEY (არაცხადად მონაწილეობს RELAX-ში) სტრ. 9) INSERT და EXTRACT-MIN პროცედურების გამოძახება ხდება ყოველი წვეროსთვის ერთხელ (წვეროების რაოდენობაა $|V|$). რადგან, ყოველი წვერო S სიმრავლეში ემატება ერთხელ, ალგორითმის მუშაობის პროცესში, ყოველი წიბოს (რომელთა რაოდენობაა $|E|$) დამუშავება მოსაზღვრე წვეროთა სიაში ხდება ერთხელ (სტრ. 8-9) ე.ი. სრულდება DECREASE-KEY-ს არა უმეტეს $|E|$ ოპერაციისა.
თუ პრიორიტეტებიანი Q რიგი რეალიზებულია როგორც მასივი, მაშინ EXTRACT-MIN ოპერაციას დასჭირდება $O(V)$ დრო; ალგორითმი ასრულებს ამ ოპერაციას $|V|$-ჯერ, ამიტომ რიგიდან ყველა ელემენტის ამოღებას დასჭირდება $O(V^2)$ დრო. ყველა დანარჩენ ოპერაციას სჭირდება $O(E)$ დრო. INSERT და DECREASE-KEY პროცედურებს სჭირდებათ $O(1)$ დრო. ალგორითმის მუშაობის დროა $O(V^2 + E^2) = O(V^2)$
თუ გრაფი ხალვათია ($|E| \ll |V|^2$), აზრი აქვს Q რიგის რეალიზებას, ორობითი გროვის საშუალებით. ორობითი გროვის აგებას დასჭირდება $O(V)$ დრო, EXTRACT-MIN ოპერაციას დასჭირდება $O(\log V)$ დრო, ასეთი ოპერაციების რაოდენობაა $|V|$, DECREASE-KEY ოპერაციას დასჭირდება $O(\log V)$ დრო, ასეთი ოპერაციების რაოდენობაა არა უმეტეს $\|$; ხოლო დეიქსტრას ალგორითმის მუშაობის სრული დრო იქნება $O((V+E)\log V) = O(E \log V)$, თუ ყველა წვერო მიღწევადია საწყისი წვეროდან. თუ პრიორიტეტებიანი Q რიგი რეალიზებულია ფიბონაჩის გროვის სახით, ალგორითმის მუშაობის დრო შეიძლება შემცირდეს $O(V \log V + E)$-მდე.

4.5 იენის ალგორითმი

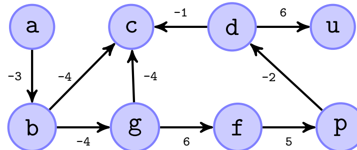
(ბელმან-ფორდის ალგორითმის მოდიფიკაცია) G გრაფის წვეროები გადავნომროთ ნებისმიერად და გრაფის წიბოთა E სიმრავლე გავყოთ ორ ნაწილად: E_f - წიბოები, რომლებიც მიმართულია ნაკლები ნომრის მქონე წვეროდან მეტი ნომრის მქონე წვეროსაკენ და E_b - წიბოები, რომლებიც მიმართულია მეტი ნომრის მქონე წვეროდან ნაკლები ნომრის მქონე წვეროსაკენ. ვთქვათ, $G_f = (V, E_f)$ და $G_b = (V, E_b)$, სადაც V გრაფის წვეროთა სიმრავლეა. ცხადია, რომ G_f და G_b გრაფები აციკლურია და ორივე მათგანი დალაგებულია ტოპოლოგიურად.
ბელმან-ფორდის ალგორითმის ციკლის ყოველ იტერაციაზე მოვახდინოთ წიბოთა რელაქსაცია შემდეგნაირად: ჯერ გადავარჩიოთ წვეროები ნომრების ზრდადობის მიხედვით და ყოველი წვეროსათვის მოხდეს მისგან გამომავალი E_f გრაფის ყველა წიბოს რელაქსაცია, შემდეგ წვეროები ნომრების კლებადობის მიხედვით და ყოველი წვეროსათვის მოხდეს მისგან გამომავალი E_b გრაფის ყველა წიბოს რელაქსაცია. ციკლის $|V|/2$ იტერაციის შემდეგ გრაფის ყველა წვეროსათვის შესრულებული იქნება $d[v] = \delta(s, v)$.

4.6 სავარჯიშოები

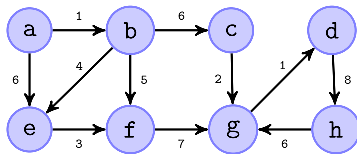
1. ბელმან-ფორდის ალგორითმით იპოვეთ უმოკლესი გზები a წვეროდან შემდეგ გრაფში:



2. შემდეგ გრაფში შეასრულეთ DAG-SHORTEST-PATHS(G,w,a):



3. დეიქსტრას ალგორითმით იპოვეთ უმოკლესი გზები a წვეროდან შემდეგ გრაფში:



4. მოიყვანეთ უარყოფითი წონის მქონე წიბოს შემცველი ორიენტირებული გრაფის მაგალითი, რომლისთვისაც დეიქსტრას ალგორითმი იძლევა არასწორ შედეგს.

თავი 5

უმოკლესი გზები წვეროთა ყველა წყვილისათვის

განვიხილოთ წონადი $G = (V, E)$ ორიენტირებული გრაფი წონითი $w : E \rightarrow R$ ფუნქციით და ვთქვათ, ჩვენი ამოცანაა წვეროთა ყოველი $u, v \in V$ წყვილისათვის ვიპოვოთ უმოკლესი გზა u-დან v-ში. ამ ამოცანის გამომავალი მონაცემები, როგორც წესი, წარმოდგენილია ცხრილის სახით, რომელშიც u სტრიქონისა და v სვეტის გადაკვეთაზე მოცემულია უმოკლესი გზის წონა u-დან v-ში. რა თქმა უნდა, ამოცანა შეიძლება გადაწყდეს, თუკი V-ჯერ გამოვიყენებთ ერთი წვეროდან უმოკლესი გზების პოვნის ალგორითმს (სათითაოდ ყველა წვეროდან, მაგრამ დეიქსტრას ალგორითმის უბრალო რეალიზაციისას, როცა პრიორიტეტებიანი რიგი რეალიზებულია როგორც მასივი, ალგორითმის მუშაობის დრო იქნება $O(V^3)$, ორობითი გროვების გამოყენებით - $O(VE \log V)$, ხოლო ფიბონაჩის გროვების შემთხვევაში - $O(V^2 \log V + VE)$. თუკი გრაფი შეიცავს უარყოფითწონიან წიბოებს, მაშინ დეიქსტრას ალგორითმის გამოყენება დაუშვებელია, ხოლო უფრო ნელი ბელმან-ფორდის ალგორითმი დახარჯავს $O(V^2 E)$ დროს (მკვრივი გრაფებისათვის - $O(V^4)$).

ერთი წვეროდან უმოკლესი გზების პოვნის ალგორითმებისგან განსხვავებით, სადაც გრაფი წარმოდგენილია მოსაზღვრე წვეროთა სიით, აქ განხილული ალგორითმები (ჯონსონის ალგორითმის გარდა) იყენებენ გრაფის წარმოდგენას მოსაზღვრეობის მატრიცის სახით. ვთქვათ, $G = (V, E)$ გრაფის წვეროები გადანომრილია, როგორც $1, 2, \ldots, |V|$. შემავალი მონაცემი იქნება მატრიცა $W = (w_{ij})$, განზომილებით $|V| \times |V|$, სადაც:

$$w_{ij} = \begin{cases} 0 & \text{თუ } i = j \\ (i,j) \text{ ორიენტირებული წიბოს წონა} & \text{თუ } i \neq j \text{ და } (i,j) \in E \\ \infty & \text{თუ } i \neq j \text{ და } (i,j) \notin E \end{cases}$$

წვეროთა ყველა წყვილისათვის უმოკლესი გზების პოვნის ალგორითმების გამომავალ მონაცემებს აქვს $|V| \times |V|$ განზომილების მქონე $D = (d_{ij})$ მატრიცის სახე, სადაც d_{ij} არის უმოკლესი გზის წონა i წვეროდან j წვერომდე. წვეროთა ყველა წყვილისათვის უმოკლესი გზების პოვნის ამოცანის ამოსახსნელად, უმოკლესი გზების წონების პოვნის გარდა, აუცილებელია ავაგოთ წინამორბედობის მატრიცა $\Pi = (\pi_{ij})$, სადაც π_{ij} იღებს NIL მნიშვნელობას, თუ $i = j$, ან გზა i წვეროდან j წვერომდე არ არსებობს; წინააღმდეგ შემთხვევაში π_{ij} არის j წვეროს მშობელი i წვეროდან რაიმე უმოკლეს გზაზე. ყოველი $i \in V$ წვეროსთვის განვსაზღვროთ $G = (V, E)$ გრაფის **წინამორბედობის ქვეგრაფი** (predecessor subgraph), $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$ სადაც:

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq NIL\} \cup \{i\} \qquad E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} \backslash \{i\}\}$$

თუ $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$ არის უმოკლესი გზების ხე, მაშინ ქვემოთ მოყვანილი პროცედურა გვაძლევს უმოკლეს გზას i წვეროდან j წვერომდე.

5.1 ფლოიდ-ვორშელის ალგორითმი

ეს ალგორითმი იყენებს დინამიკური პროგრამირების მეთოდს. იგი მუშაობს უარყოფითწონებიანი წიბოების შემცველი გრაფებისთვისაც, რომლებიც არ შეიცავენ უარყოფითწონიან ციკლებს.

Algorithm 15: Print All Pairs Shortest Path

Input: წინამორბედობის მატრიცა Π, i და j წვეროები
Output: ბეჭდავს გზას ი-დან ჯ-მდე

```
1 PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j) :
2     if i == j :
3         print(i);
4     elif Π[i][j] == NIL :
5         print('i− დან j-ში გზა არ არსებობს!);
6     else:
7         PRINT-PATH(Π, i, Π[i][j]);
8         print(j);
```

5.1.1 უმოკლესი გზის სტრუქტურა

ფლოიდ-ვორშელის ალგორითმში განიხილება უმოკლესი გზის **შუალედური** (intermediate) **წვეროები**. მარტივი $p =< v_1, v_2, \dots, v_t >$ გზის **შუალედური წვერო** ეწოდება $v_2, v_3, \dots, v_{t-1}$ წვეროებიდან ნებისმიერს.
ჩავთვალოთ, რომ G გრაფი შედგება $V = \{1, 2, \dots, n\}$, წვეროებისგან. რაიმე $k \leq n$-თვის განვიხილოთ წვეროთა ქვესიმრავლე $\{1, 2, \dots, k\}$. ნებისმიერი $i, j \in V$ წვეროთა წყვილისათვის განვიხილოთ ყველა გზა i-დან j-ში, რომელთა შუალედური წვეროები ეკუთვნიან $\{1, 2, \dots, k\}$, სიმრავლეს. ვთქვათ p-მინიმალური წონის გზაა ყველა ასეთ გზას შორის. ის იქნება მარტივი, რადგან გრაფში არაა უარყოფითწონიანი ციკლი. ფლოიდ-ვორშელის ალგორითმში გამოიყენება ურთიერთკავშირი p გზასა და i წვეროდან j წვეროში იმ უმოკლეს გზებს შორის, რომელთა შუალედური წვეროები ეკუთვნიან $\{1, 2, \dots, k-1\}$ სიმრავლეს. ეს ურთიერთკავშირი დამოკიდებულია იმაზე, არის თუ არა k შუალედური წვერო p გზისთვის.

- თუ k წვერო არ წარმოადგენს შუალედურ წვეროს p გზისათვის, მაშინ p გზის ყველა შუალედური წვერო მოთავსებულია $\{1, 2, \dots, k-1\}$ სიმრავლეში. ამრიგად, უმოკლესი გზა i წვეროდან j წვეროში ყველა შუალედური წვეროთი $\{1, 2, \dots, k-1\}$ სიმრავლიდან, წარმოადგენს, ამავე დროს, უმოკლეს გზას i წვეროდან j წვეროში ყველა შუალედური წვეროთი $\{1, 2, \dots, k\}$ სიმრავლიდან.
- თუ k წვერო შუალედურია p გზისათვის, მაშინ იგი ჰყოფს p-ს ორ p_1 და p_2 ნაწილებად. **ლემა** 4.1-ის თანახმად, p_1 არის უმოკლესი გზა i-დან k-მდე, ყველა შუალედური წვეროთი $\{1, 2, \dots, k\}$ სიმრავლიდან. რადგან k არ არის p_1 გზის შუალედური წვერო, ამიტომ p_1 არის უმოკლესი გზა i-დან k-მდე, ყველა შუალედური წვეროთი $\{1, 2, \dots, k-1\}$ სიმრავლიდან. ანალოგიურად, p_2 არის უმოკლესი გზა k-დან j-მდე ყველა შუალედური წვეროთი $\{1, 2, \dots, k-1\}$ სიმრავლიდან.

5.1.2 წვეროთა ყველა წყვილისათვის უმოკლესი გზების შესახებ ამოცანის რეკურსიული ამოხსნა

აღვნიშნოთ $d_{ij}^{(k)}$-თი უმოკლესი გზის წონა i წვეროდან j წვეროში შუალედური წვეროებით $\{1, 2, \dots, k\}$, სიმრავლიდან. თუ $k = 0$, მაშინ შუალედური წვეროები საერთოდ არა გვაქვს. ასეთი გზა შეიცავს არა უმეტეს ერთ წიბოს, ამიტომ $d_{ij}^{(0)} = w_{ij}$. საზოგადოდ:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{თუ } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{თუ } k \geq 1 \end{cases}$$

რადგან, ნებისმიერი გზის ყველა შუალედური წვერო ეკუთვნის $\{1, 2, \dots, n\}$ სიმრავლეს, მატრიცა $D^{(n)} = (d_{ij}^{(n)})$ შეიცავს საძიებელ მნიშვნელობებს, ანუ $d_{ij}^{(n)} = \delta(i, j)$ წვეროთა ყველა წყვილისთვის.

5.1.3 უმოკლესი გზების წონების გამოთვლა

დავწეროთ პროცედურა, რომელიც გამოთვლის უმოკლესი გზების წონებს $d_{ij}^{(k)}$ $k = 1, \dots, n$ მნიშვნელობების თანმიმდევრული ჰოვნით. მისთვის შემავალი მონაცემი იქნება $n \times n$ $n = |V|$ ზომის W მატრიცა, (იხ. (1)) რომელშიც მოცემულია გრაფის წიბოთა წონები, ამ ალგორითმით გამოითვლება D^k, $k = 1, \dots, n$ მატრიცები, გამომავალი მონაცემი კი იქნება უმოკლესი გზების წონათა $D^{(n)}$ მატრიცა.

Algorithm 16: Floyd Warshall All Pairs Shortest Paths

Input: გრაფის ინციდენტობის მატრიცა W რომლის i სტრიქონისა და j სვეტის გადაკვეთაზე წერია (i, j) წიბოს წონა

Output: მატრიცა რომლის i სტრიქონისა და j სვეტის გადაკვეთაზე წერია i წვეროდან j წვერომდე უმოკლესი გზის წონა

```
1 FLOYD-WARSHALL(W) :
2   n = |V|;
3   D^(0) = W;
4   for k=1; k<=n; k++ :
5     for i=1; i<=n; i++ :
6       for j=1; j<=n; j++ :
7         d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1));
8   return D^(n)
```

შევნიშნოთ, რომ:

$$d_{ik}^{(k)} = min(d_{ik}^{(k-1)}, d_{ik}^{(k-1)} + 0) = d_{ik}^{(k-1)}$$

$$d_{kj}^{(k)} = min(d_{kj}^{(k-1)}, 0 + d_{kj}^{(k-1)} + 0) = d_{kj}^{(k-1)}$$

ამიტომ, $D^{(k)}$ მატრიცის k-ური სვეტი და k-ური სტრიქონი ემთხვევა $D^{(k-1)}$ მატრიცის k-ურ სვეტს და k-ურ სტრიქონს.

5.1.4 უმოკლესი გზის აგება

უმოკლესი გზების წონათა გარდა, ხშირად საჭიროა თავად ამ გზის პოვნაც. ფლოიდ-ვორშელის ალგორითმში უმოკლესი გზების პოვნის მრავალი მეთოდი არსებობს. ერთ-ერთი მათგანია D მატრიცის (რომელიც შეიცავს უმოკლესი გზების წონებს) გამოთვლა და მისი საშუალებით Π წინამორბედობის მატრიცის აგება. მაგრამ უფრო მოსახერხებელია, გზები გამოვთვალოთ ფლოიდ-ვორშელის ალგორითმის პარალელურად. უნდა გამოვითვალოთ მატრიცები: $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, სადაც $\Pi = \Pi^{(n)}$, ხოლო π_{ij} ელემენტი განისაზღვრება როგორც j წვეროს მშობელი უმოკლეს გზაზე i-დან j-ში შუალედური წვეროებით $\{1, 2, \ldots, k\}$ სიმრავლიდან. თუ $k = 0$, მაშინ უმოკლესი გზები i-დან j-ში არ შეიცავენ შუალედურ წვეროებს, ამრიგად:

$$\pi_{ij}^{(0)} = \begin{cases} NIL & \text{თუ } i = j \text{ ან } w_{ij} = \infty \\ i & \text{თუ } i \neq j \text{ და } w_{ij} < \infty \end{cases}$$

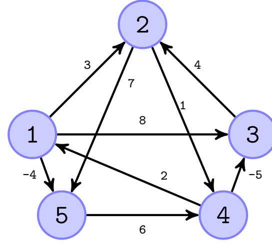
თუ $k \geq 1$-თვის, უმოკლესი გზა i-დან j-ში გადის k წვეროზე $(k \neq j)$, მაშინ j წვეროს მშობელი დაემთხვევა ამავე წვეროს მშობელს უმოკლეს გზაზე k წვეროდან, შუალედური წვეროებით $\{1, 2, \ldots, k-1\}$ სიმრავლიდან. ხოლო თუ გზა არ გადის k-ზე, მაშინ ვირჩევთ j წვეროს იმავე მშობელს, რაც იყო უმოკლეს გზაზე i წვეროდან შუალედური წვეროებით $\{1, 2, \ldots, k-1\}$, სიმრავლეში.
მაშასადამე $k \geq 1$-თვის:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{თუ } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{თუ } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

ამ ფორმულების დამატება FLOYD-WARSHALL პროცედურისათვის რთული არაა. სურ. 5.1-ზე ნაჩვენებია D და Π მატრიცების შევსება ფლოიდ-ვორშელის ალგორითმის მიხედვით ამავე სურათზე გამოსახული გრაფისათვის. **ფლოიდ-ვორშელის ალგორითმის მუშაობის დრო.** მუშაობის დრო განისაზღვრება სამჯერ ჩადგმული for ციკლით (სტრ. 4-7). რადგან სტრ. 7-ის შესრულებას სჭირდება $O(1)$ დრო, ალგორითმი ასრულებს მუშაობას $\Theta(n^3)$ დროში.

5.2 ორიენტირებული გრაფის ტრანზიტული ჩაკეტვა

ორიენტირებული გრაფის ტრანზიტული ჩაკეტვის ამოცანა მდგომარეობს შემდეგში: მოცემულია $G = (V, E)$ გრაფი $1, 2, \ldots, n$ წვეროებით. საჭიროა ნებისმიერი $i, j \in V$ წვეროებისათვის გაირკვეს, არსებობს თუ არა გრაფში



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

ნახ. 5.1

გზა i წვეროდან j წვერომდე. ორიენტირებული G გრაფის **ტრანზიტული ჩაკეტვა** (transitive closure) ეწოდება $G^* = (V^*, E^*)$ გრაფს, სადაც $E^* = \{(i, j) : G\text{-ში } \exists \text{გზა } i\text{-დან } j\text{-ში}\}$.

გრაფის ტრანზიტული ჩაკეტვის პოვნის ერთ-ერთი გზაა $\Theta(n^3)$ დროში, მივანიჭოთ ყველა წიბოს 1-ის ტოლი მნიშვნელობა და შევასრულოთ ფლოიდ-ვორშელის ალგორითმი. თუ არსებობს გზა i-დან j-ში, მაშინ d_{ij} იქნება n-ზე ნაკლები, ხოლო თუ ასეთი გზა არ არსებობს, მაშინ $d_{ij} = \infty$.

დროისა და მანქანური მეხსიერების ეკონომიის მიზნით, უმჯობესია ფლოიდ-ვორშელის ალგორითმში არითმეტიკული ოპერაციები min და $+$, შევცვალოთ ლოგიკური ოპერაციებით OR (ან) და AND (და). უფრო ზუსტად, $t_{ij}^{(k)}$ ჩავთვალოთ 1-ის ტოლად, თუ G გრაფში არსებობს გზა i-დან j-ში, რომლის ყველა შუალედური წვერო მოთავსებულია $\{1, 2, \ldots, k\}$ სიმრავლეში და ჩავთვალოთ 0-ის ტოლად, თუკი ასეთ გზა არ არსებობს. (i, j) წიბო ეკუთვნის $G*$ ტრანზიტულ ჩაკეტვას, მაშინ და მხოლოდ მაშინ, როცა $t_{ij}^{(k)} = 1$, ე.ი.

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{თუ } i \neq j \text{ და } (i,j) \notin E \\ 1 & \text{თუ } i = j \text{ ან } (i,j) \in E \end{cases}$$

ხოლო თუ $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \; OR \; (t_{ik}^{(k-1)} \; AND \; t_{kj}^{(k-1)})$$

ამ თაფარდობებზე დაყრდნობით ალგორითმი თანმიმდევრობით გამოითვლის $T^{(k)} = (t_{ij}^{(k)})$ მატრიცას - $k = 1, 2, \ldots, n$-თვის:

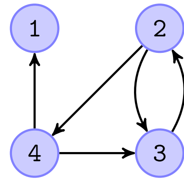
Algorithm 17: Transitive Closure

Input: ორიენტირებული გრაფი $G = (V, E)$

Output: მატრიცა რომლის i სტრიქონისა და j სვეტის გადაკვეთაზე წერია არსებობს თუ არა გზა i წვეროდან j წვერომდე

```
1  TRANSITIVE-CLOSURE(G) :
2      n = |V|;
3      for i=1; i<=n; i++ :
4          for j=1; j<=n; j++ :
5              if i == j or (i,j) ∈ E :
6                  t_ij^(0) = 1;
7              else:
8                  t_ij^(0) = 0;
9      for k=1; k<=n; k++ :
10         for i=1; i<=n; i++ :
11             for j=1; j<=n; j++ :
12                 t_ij^(k) = t_ij^(k-1) or t_ik^(k-1) and t_kj^(k-1);
13     return T^(n)
```

სურ. 5.2-ზე მოცემულია გრაფი და ალგორითმის მუშაობის პროცესი ამ გრაფისათვის:



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

ნახ. 5.2

TRANSITIVE-CLOSURE-ს მუშაობის დრო ფლოიდ-ვორშელის ალგორითმის მსგავსად არის $\Theta(n^3)$, მაგრამ უფრო ეფექტურია რიგ შემთხვევებში, რადგან ლოგიკური ოპერაციები ზოგ კომპიუტერზე უფრო სწრაფად სრულდება, ვიდრე არითმეტიკული ოპერაციები მთელ რიცხვებზე, ხოლო ლოგიკურ ცვლადებს ნაკლები მეხსიერება უჭირავთ.

5.3 ჯონსონის ალგორითმი ხალვათი გრაფებისათვის

ჯონსონის ალგორითმი პოულობს უმოკლეს გზებს წვეროთა ყველა წყვილისათვის $O(V^2 \log V + VE)$ დროში და ამიტომ ხალვათი გრაფებისათვის უფრო ეფექტურია, ვიდრე ფლოიდ-ვორშელის ალგორითმი. ჯონსონის ალგორითმი ან იძლევა უმოკლესი გზების წონათა მატრიცას ან იტყობინება, რომ გრაფში არის უარყოფითწონიანი ციკლი. ეს ალგორითმი იყენებს დეიქსტრასა და ბელმან-ფორდის ალგორითმებს.
ჯონსონის ალგორითმი დაფუძნებული წონათა ცვლილების (reweighting) იდეაზე. თუ გრაფის ყველა წიბოს წონა არაუარყოფითია, მაშინ დეიქსტრას ალგორითმის გამოყენებით თითოეული წვეროსათვის შეგვიძლია ვიპოვოთ უმოკლესი გზები წვეროთა ყველა წყვილისათვის. თუკი გრაფში არის უარყოფითწონიანი წიბოები და არ არის უარყოფითწონიანი ციკლი, შეგვიძლია ასეთი შემთხვევა დავიყვანოთ არაუარყოფითწონიანი წიბოების შემთხვევაზე w წონითი ფუნქციის შეცვლით ახალი $\hat{w}$ ფუნქციით. ამასთან უნდა შესრულდეს შემდეგი თვისებები:

1. უმოკლესი გზები არ იცვლება: წვეროთა ნებისმიერი $u, v \in V$ წყვილისათვის უმოკლესი გზა u-დან v-ში w წონითი ფუნქციის მიხედვით წარმოადგენს უმოკლეს გზას $\hat{w}$ წონითი ფუნქციის მიხედვითაც და პირიქით

2. ნებისმიერი $u, v \in V$ წყვილისათვის ახალი $\hat{w}(u, v)$ არაუარყოფითია

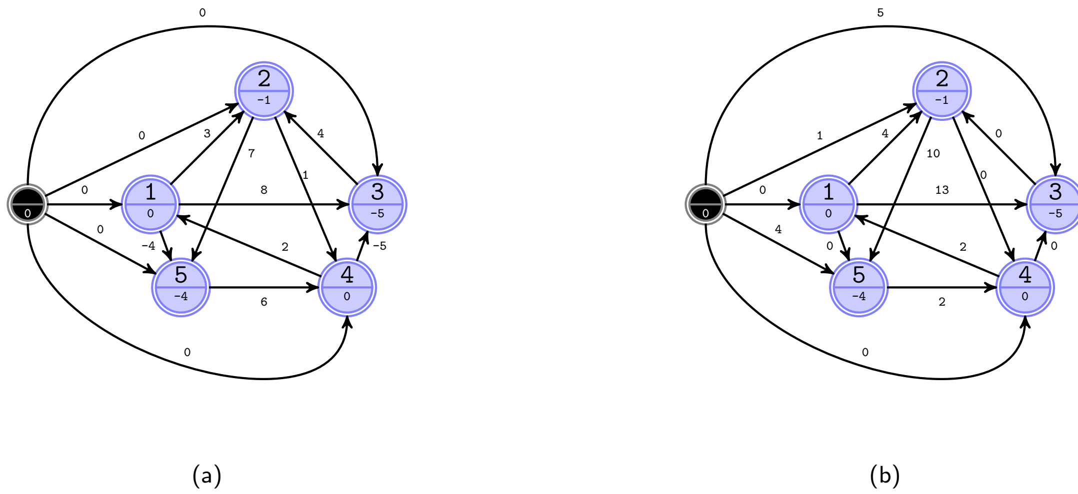
ლემა 5.1. (წონათა ცვლილება არ ცვლის უმოკლეს გზებს). ვთქვათ $G = (V, E)$ წონადი ორიენტირებული გრაფია წონითი $w : E \to R$ ფუნქციით. ვთქვათ $h : V \to R$ გრაფის წვეროებზე განსაზღვრული ფუნქციაა ნამდვილი მნიშვნელობებით. ყოველი $(u, v) \in E$ წიბოსთვის განვიხილოთ ახალი წონითი ფუნქცია $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$, ვთქვათ, $p =< v_0, v_1, \ldots, v_k >$ რაიმე გზაა v_0 წვეროდან v_k წვეროში მაშინ:

1. p გზა წარმოადგენს უმოკლეს გზას, w წონითი ფუნქციის შემთხვევაში მაშინ და მხოლოდ მაშინ, როცა ის იქნება უმოკლესი გზა $\hat{w}$ წონითი ფუნქციისთვის. ანუ შემდეგი ორი ტოლობა $w(p) = \delta(v_0, v_k)$, $\hat{w}(p) = \delta(v_0, v_k)$ ტოლფასია.

2. G გრაფი შეიცავს უარყოფითწონიან ციკლს w წონითი ფუნქციის შემთხვევაში მაშინ და მხოლოდ მაშინ, როცა ის შეიცავს უარყოფითწონიან ციკლს $\hat{w}$ წონითი ფუნქციისთვის.

ახლა უნდა შევარჩიოთ h ფუნქცია ისე, რომ ყოველი (u, v) წიბოსთვის შეცვლილი $\hat{w}(u, v)$ წონები არაუარყოფითი იყოს (თვისება 2). ამისათვის ავაგოთ ახალი $G' = (V', E')$ გრაფი $G = (V, E)$ გრაფისთვის ერთი წვეროს დამატებით, რომლიდანაც გრაფის ყველა სხვა წვეროსაკენ მიმართული იქნება ნულოვანი წონის წიბოები. $V' = V \cup \{s\}$ $E' = E \cup \{(s, v) : v \in V\}$ $w(s, v) = 0$.
აღვნიშნოთ, რომ რადგან s წვეროში არ შედის არც ერთი წიბო, ამიტომ იგი შედის G' გრაფის მხოლოდ იმ უმოკლეს გზებში, რომლებიც ამ წვეროდან იღებს სათავეს. ცხადია, რომ ახალ G' გრაფში უარყოფითწონიანი ციკლი არ იქნება მაშინ და მხოლოდ მაშინ, თუ ასეთი ციკლი არ არსებობდა G გრაფში.

დავუშვათ, რომ G და G' გრაფები არ შეიცავენ უარყოფითწონიან ციკლებს. ნებისმიერი $v \in V'$-სათვის განვსაზღვროთ $h(v) = \delta(s, v)$. სამკუთხედის უტოლობიდან გამომდინარე (იხ. **ლექცია 6-7**) ნებისმიერი $(u, v) \in E'$ წიბოსათვის სრულდება $h(v) \leq h(u)+w(u, v)$ უტოლობა, რომელიც შეიძლება ასე გადავწეროთ: $w(u, v)+h(u)-h(v) \geq 0$. ეს კი ნიშნავს, რომ $\hat{w}(u, v)$ ახალი წონითი ფუნქცია არაუარყოფითია.



ნახ. 5.3

სურ. 5.3-ზე გამოსახულია G გრაფის შესაბამისი G' გრაფი. დამხმარე s წვერო აღნიშნულია შავად. ყოველი წვეროს შიგნით ჩაწერილია მნიშვნელობა $h(v) = \delta(s, v)$. ბ) ნახაზზე ყოველ (u, v) წიბოს მიწერილი აქვს ახალი წონა - $w(u, v) + h(u) - h(v)$.

ჯონსონის ალგორითმში გამოიყენება ბელმან-ფორდის და დეიქსტრას ალგორითმები. გრაფი წარმოდგენილია მოსაზღვრე წვეროთა სიით. ეს ალგორითმი აბრუნებს $D = d_{ij}$ მატრიცას, $d_{ij} = \delta(i, j)$, განზომილებით $|V| \times |V|$, ან შეტყობინებას, რომ შემავალი გრაფი შეიცავს უარყოფითწონიან ციკლს. წვეროები გადანომრილია 1-დან V-მდე.

ალგორითმის მუშაობის დრო დამოკიდებულია დეიქსტრას ალგორითმში პრიორიტეტებიანი რიგის რეალიზაციაზე. თუ ის რეალიზებულია ფიბონაჩის გროვის სახით, ჯონსონის ალგორითმის მუშაობის დროა $O(V^2 \log V + VE)$, ხოლო უფრო უბრალო რაელიზაციისას - $O(VE \log V)$, რაც ხალვათი გრაფებისთვის, ასიმპტოტურად უკეთესია ფლოიდ-ვორშელის ალგორითმზე.

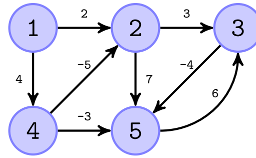
Algorithm 18: Johnson (All Pairs Shortest Paths)

Input: ორიენტირებული გრაფი $G = (V, E)$ და წონითი ფუნქცია $w : E \to R$
Output: უმოკლესი გზების წონათა მატრიცა ან FALSE, თუ გრაფი შეიცავს უარყოფითი წონის ციკლს

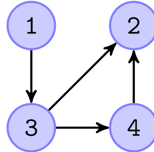
```
1  JOHNSON(G, w) :
2      SevqmnaT G′ = (V′, E′) grafi; //V′ = V ∪ {s}
3                                     //E′ = E ∪ {(s, v) : ∀v ∈ V}
4                                     //w(s, v) = 0 : ∀v ∈ V}
5      if BELLMAN-FORD(G′, w, s) == FALSE :
6          print(' გრაფი შეიცავს უარყოფითწონიან ციკლს!');
7          return FALSE;
8      else:
9          for ∀v ∈ V′ :
10             h[v] = δ(s, v); // გამოითვლება BELLMAN-FORD- ით
11             for ∀(u, v) ∈ E′ :
12                 ŵ(u, v) = w(u, v) + h[u] − h[v];
13             for ∀u ∈ V :
14                 DIJKSTRA(G, ŵ, u); // გამოვთვალოთ δ̂(u, v) : ∀v ∈ V
15                 for ∀v ∈ V :
16                     d[u][v] = δ̂(u, v) + h[v] − h[u];
17         return D;
```

5.4 სავარჯიშოები

1. ფლოიდ-ვორშელის ალგორითმით იპოვეთ და ააგეთ უმოკლესი გზები შემდეგ გრაფში:



2. ამოხსენით ტრანზიტული ჩაკეტვის ამოცანა შემდეგი გრაფისთვის:



3. რა იქნება იმ ორიენტირებული V - წვეროიანი გრაფის ტრანზიტული ჩაკეტვა, რომელიც მხოლოდ ციკლისგან შედგება?

4. რამდენ წიბოს შეიცავს ორიენტირებული V - წვეროიანი გრაფის ტრანზიტული ჩაკეტვა, რომელიც შედგება მხოლოდ მარტივი ორიენტირებული გზისგან?

თავი 6

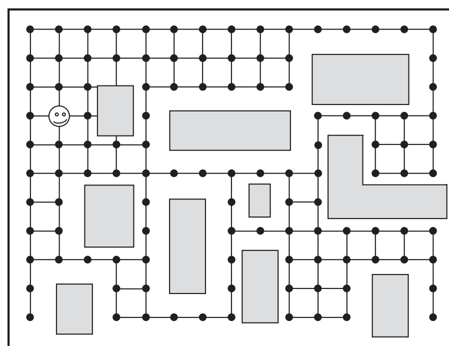
ამოცანათა გრაფებზე გადატანის მაგალითები

ამოცანათა სცორად ჩამოყალიბება და გრაფებზე გადატანა (მოდელირება) უმნიშვნელოვანეს როლს თამაშობს მათ გადაწყვეტაში - სწორად დასმული ამოცანა ნახევარი ამოხსნის ტოლფასია. აქამდე გრაფებთან დაკავშირებულ ძირითად ამოცანებსა და განსაზღვრებებს განვიხილავდით, ახლა კი გადავიდეთ ამოცანათა მოდელირების მაგალითებზე და მოვიყვანოთ ის უმნიშვნელოვანესი ამოცანები, რომელთა გადაჭრაც ხშირ შემთხვევაში მრავალი სხვა პრაქტიკული ამოცანის დაძლევაში მოგვეხმარება.
ქვემოთ მოყვანილი მაგალითებით ნათლად უნდა ჩანდეს, თუ როგორ შეიძლება ამა თუ იმ ამოცანის გრაფების მეშვეობით გადაჭრა. ხშირ შემთხვევაში გამოვიყენებთ იმ ფაქტს, რომ ამოცანის მონაცემთა ელემენტების სიმრავლეზე *მიმართებების* განმარტება შეიძლება (იმის გათვალისწინებით, თუ რა დამოკიდებულებაა ამ ელემენტებს შორის), ხოლო მიმართებების გამოსახვა გრაფთა მეშვეობით ადვილად შეიძლება.

შენიშვნა: ამოცანის პირობის წაკითხვის შემდეგ, სანამ კითხვას გააგრძელებთ, თქვენ თვითონ დაფიქრდით საკითხზე, თუ როგორ შეიძლება მისი გადატანა გრაფებზე.

6.1 მოძრაობა ვიდეო თამაშებში.

წარმოიდგინეთ, რომ გვინდა ვამოძრაოთ ვიდეო თამაშის გმირი ოთახში, რომელშიც საგნებია განთავსებული. როგორ შეიძლება ოპტიმალური გზის გამოანგარიშება?
ამ ამოცანის ამოხსნისას ბუნებრივად გრაფებში უმოკლესი გზის პოვნის ასოციაცია ჩნდება. მაგრამ როგორი უნდა იყოს გრაფი? პირველი, რაც თავში შეიძლება მოგვივიდეს, შემდეგი იდეაა: ოთახის სურათს დავადოთ ბადე, შემდეგ ამოვყაროთ ის წერტილები (მათთან მიერთებულ წიბოებთან ერთად), რომლებიც საგნების ნახატებს ემთხვევა. მივიღებთ გრაფს, რომლის წიბოებს შორის მანძილი ერთის ტოლად შეიძლება მივიჩნიოთ (ანუ გრაფი არ იქნება შეწონილი).



ნახ. 6.1: მოძრაობის ბადე

რა თქმა უნდა, შეგვეძლო სხვა გრაფის შექმნაც, რომელიც უფრო ეფექტურად გადაჭრიდა ამ ამოცანას (მაგალითად, არაა აუცილებელი სულ მართი კუთხე გვქონდეს - ზოგჯერ შეიძლება ზემოთ მოყვანილი ბადის დიაგონალებზე გასვლა, ან ისეთ ადგილებში, სადაც ბადის ორი პარალელური ხაზი გადის ერთის აღება და ა.შ. ამ შემთხვევებში გრაფები უკვე შეწონილი უნდა იყოს). ამას გარდა, არსებობს გეომეტრიული ალგორითმები, რომლებიც ანალოგიური ამოცანებისათვის უფრო ეფექტურად გადაჭრიდა უმოკლესი გზის საკითხს, მაგრამ ამ სახის ალგორითმის იმპლემენტაცია გაცილებით უფრო მარტივია და შედეგიც არ იქნება საგრძნობლად უარესი.

6.2 გენეტური კოდის აგება.

ამ ამოცანაში მოცემული გვაქვს დნმ კოდის ნაწილები $\Phi = (\phi_1, \phi_2, ..., \phi_n)$, რომლებიც ექსპერიმენტების შედეგად იქნა მიღებული. ყოველი $f \in \Phi$ ფრაგმენტისათვის ვიპოვნით ისეთ ელემენტებს, რომლებიც უნდა განთავსდნენ f ფრაგმენტის მარჯვნივ (ან, შესაბამისად, მარცხნივ). როგორც წესი, იარსებებს აგრეთვე ისეთი (ერთი ან რამოდენიმე) ელემენტი, რომლის განთავსებაც ორივე მხარეს შეიძლება. აღსანიშნავია, რომ განლაგების წესი ტრანზიტულია: თუ f_1 ფრაგმენტი f_2 ფრაგმენტის მარცხნივ და ეს კი თავის თავად f_3 ფრაგმენტის მარცხნივ უნდა განთავსდეს, მაშინ f_1 უნდა აღმოჩნდეს f_3 ნაწილის მარცხნივ.
გენეტური კოდის აგების ამოცანა იმაში მდგომარეობს, რომ ვიპოვნოთ Φ მიმდევრობის *ყველა* ელემენტისაგან შემდგარი ისეთი მიმდევრობა, რომელიც ზედა პირობებს აკმაყოფილებს. სხვა სიტყვებით რომ ვთქვათ, უნდა ვიპოვნოთ Φ მიმდევრობის ისეთი პერმუტაცია $(\phi_{i_1}, ..., \phi_{i_n})$, რომ თუ $k < l$, მაშინ ზემოთ მოყვანილი წესების თანახმად ϕ_{i_k} ფრაგმენტი *უნდა* იდგეს ϕ_{i_l} ფრაგმენტის მარცხნივ (შესაბამისად, ϕ_{i_l} ფრაგმენტი *უნდა* იდგეს ϕ_{i_k} ფრაგმენტის *მარჯვნივ*).

ამ ამოცანის გადაჭრა შემდეგნაირად შეიძლება: შევადგინოთ მიმართება

$$R = \{(\phi_i, \phi_j) | \phi_i \text{ ფრაგმენტი უნდა განთავსდეს } \phi_j \text{ ფრაგმენტის } \textit{მარცხნივ}\}.$$

ცხადია, რომ ეს მიმართება შექმნის (ერთ ან რამოდენიმე) მიმართულ აციკლურ გრაფს, რომლის ტოპოლოგიური დალაგება საძიებო მიმდევრობას მოგვცემს.

სავარჯიშო 6.1: დაამტკიცეთ, რომ ზემოთ მოყვანილი წესით მიმართულ *აციკლურ* გრაფს მივიღებთ.

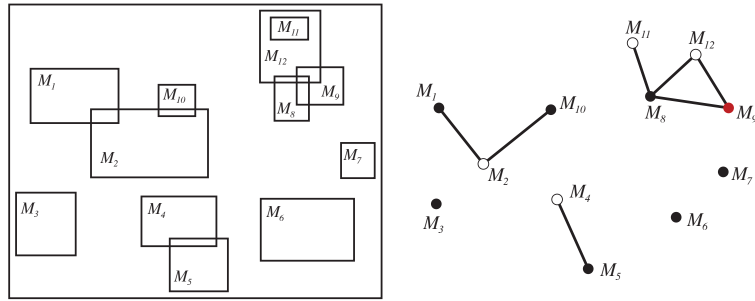
სავარჯიშო 6.2: დაამტკიცეთ, რომ ამ აციკლური გრაფების ტოპოლოგიური დალაგება მართლაც გენეტური კოდის დასაშვებ მიმდევრობას მოგვცემს (Φ მიმდევრობის ისეთ პერმუტაციას $(\phi_{i_1}, ..., \phi_{i_n})$, რომ თუ $k < l$, მაშინ ϕ_{i_k} ფრაგმენტი *შეიძლება* იდგეს ϕ_{i_l} ფრაგმენტის მარცხნივ).

6.3 ობიექტების დაჯგუფება.

გრაფიკულ ამოცანებში ხშირად საჭიროა ხოლმე გრაფიკული ობიექტების (მაგ. მართკუთხედების) ისეთი დაჯგუფება, რომ ერთ ჯგუფში მყოფი ობიექტები ერთმანეთს არ კვეთდნენ (სხვადასხვა ჯგუფში მოხვედრილი ობიექტები ერთმანეთს შეიძლება კვეთდნენ).
ამ ამოცანის გადასაჭრელად ყოველ ობიექტს გრაფის წვერო შევუსაბამოთ. თუ ორი ობიექტი ერთმანეთს კვეთს, მაშინ შესაბამისი წვეროები წიბოთი შევაერთოთ. ცხადია, გრაფიკულ ობიექტთა ყოველი გაერთიანება ასე შექმნილი გრაფის დამოუკიდებელ წვეროთა სიმრავლეა.
გრაფის წვეროების შეღებვის ამოცანა, რომელიც ზემოთ გვქონდა აღწერილი, სწორედ ასეთ იზოლირებულ წერტილთა სიმრავლეს (და, შესაბამისად, არაგადამკვეთ ობიექტთა გაერთიანებას) მოგვცემს. გრაფის შეღებვაში გამოყენებული ფერების რაოდენობის მინიმიზაცია კი მინიმალური რაოდენობის გაერთიანებებს მოგვცემს.

ზემოთ მოყვანილი ნახაზის მაგალითში ობიექტები სამ კლასად შეიძლება დავაჯგუფოთ:

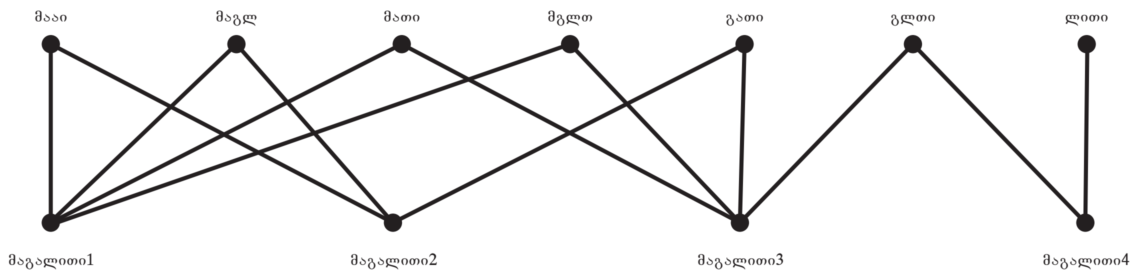
$$K_1 = \{M_1, M_3, M_5, M_6, M_7, M_8, M_{10}\},\ K_2 = \{M_2, M_4, M_{11}, M_{12}\},\ K_3 = \{M_9\}.$$



ნახ. 6.2: გეომეტრიული ობიექტები და მათი შესაბამისი (სამად შეღებილი) გრაფი

6.4 სიტყვათა შემოკლება

ზოგჯერ საჭიროა ხოლმე დიდ მონაცემთა ბაზაში მონაცემთა სახელების სიგრძის შემცირება. მაგალითად, თუ გვაქვს გრძელ სიტყვათა სიმრავლე (დავუშვათ, რამოდენიმე ასეული 32 სიმბოლოსაგან შემდგარი), მათი სიგრძის შემცირება შეიძლება (მაგალითად 8 სიმბოლოიანზე) ისე, რომ სხვადასხვა სიტყვა ისევე სხვადასხვა დარჩეს. ამ შემთხვევაში პირველი 8 ასოს აღება არ გამოდგება, რადგან „მაგალითი1" და „მაგალითი2" განსხვავებული აღარ იქნება. რა წესით უნდა შევამოკლოთ სიტყვები ისე, რომ შედეგები ერთმანეთს არ დაემთხვას?
ყოველ შესამოკლებელ სიტყვას w_i შევუსაბამოთ გრაფის ერთი წვერო v_i. შემდეგ ყოველი w_i სიტყვისათვის შევქმნათ შესაძლო შემოკლებული სიტყვები $w'_{i,j}$ და შევუსაბამოთ წვეროები $v'_{i,j}$. შემდეგ გავავლოთ წიბოები $v_i, v'_{i,j}$ ყველა შესაძლო i და j პარამეტრებისათვის (ყოველ სიტყვასა და მის შესაძლებელ შემოკლებას შორის).



ნახ. 6.3: შემოკლების გრაფი

ცხადია, რომ თუ ამ გრაფში ავიღებთ ისეთ წიბოებს, რომლებიც ერთმანეთთან არ იქნება დაკავშირებული (იზოლირებულ წიბოებს), ცალსახა შემოკლებების სიის შედგენას შევძლებთ. ასე, ნახ. 6.3-ში მოყვანილ მაგალითში შეიძლება ავიღოთ წიბოები {(მაგალითი1, მაგლ), (მაგალითი2, მააი), (მაგალითი3, გათი), (მაგალითი4, ლითი)}, რაც ერთ-ერთ შესაძლებელ შემოკლების სიას მოგვცემს.

6.5 გაყალბების აღმოჩენა.

ამ ამოცანაში საჭიროა გამყალბებელთა დოკუმენტების დადგენა. ხშირად ისე ხდება ხოლმე, რომ გამყალბებლები მათ მიერ შექმნილ საბუთებს გზავნიან (ბანკებში, საგადასახადო სისტემებში გადასახადების ასანაზღაურებლად, ბენზოგასამართ სადგურებში ან სხვა დაწესებულებებში გაყალბებული ვაუჩერებით საქონლის მისაღებად და ა.შ.), რომლებიც იდენტური არაა, მაგრამ გარკვეული თვალსაზრისით *ერთმანეთის მსგავსია*. როგორ შეიძლება მათი აღმოჩენა?
პირველ რიგში დასადგენია, როგორი საბუთები ითვლება *მსგავსად* (სხვადასხვა ამოცანისათვის ეს სხვადასხვა შეიძლება იყოს, როგორც მაგ. მსგავსი ნომრები, ფორმები, მისამართები, სახელები, გვარები და ა.შ.).
ყოველ საბუთს გრაფის ერთი წვერო შევუსაბამოთ და ორ წვეროს შორის წიბო გავავლოთ, თუ შესაბამისი საბუთები მსგავსია. ცხადია, რომ თუ ასეთ გრაფში ვიპოვნით წვეროთა სიმრავლეს, რომელიც *ბევრი* წიბოთი

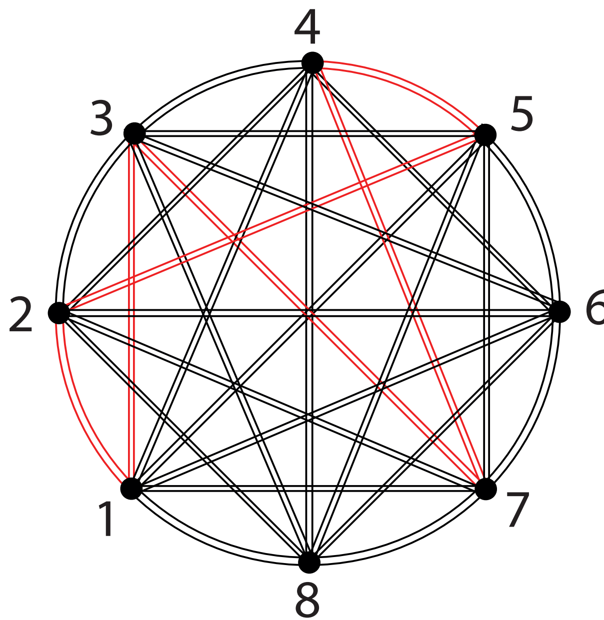
იქნება შეერთებული (მაგალითად ყველა ყველასთან - სრული ქვეგრაფი), შესაბამისი საბუთების უფრო დეტალური შესწავლა შეიძლება ღირდეს. ამ ამოცანის გადასაწყვეტად გრაფში მაქსიმალური სრული ქვეგრაფის ამორჩევის ალგორითმები შეიძლება გამოგვადგეს.

6.6 ეკონომიკური ამოცანები

ეკონომიკური ამოცანების გრაფებზე გადატანის ერთ-ერთ მაგალითად შეიძლება ვალუტის გადაცვლის პროცედურა განვიხილოთ.
ნებისმიერი ორი ქვეყნის ფულად ერთეულს შორის გაცვლითი კურსია დადგენილი, რომლის მაგალითიც შემდგომ ცხრილშია წარმოდგენილი.

		USD	EUR	JPY	GBP	CHF	CAD	AUD	HKD
1.	USD	–	1.2329	0.0093	1.3881	1.0547	0.7776	0.7866	0.127
2.	EUR	0.8111	–	0.0076	1.1259	0.8555	0.6307	0.6380	0.1035
3.	JPY	107.1400	132.0929	–	148.7210	113.0050	83.3126	84.2763	13.6653
4.	GBP	0.7204	0.8882	0.0067	–	0.7598	0.5602	0.5667	0.0919
5.	CHF	0.9481	1.1689	0.0088	1.3161	–	0.7372	0.7458	0.1209
6.	CAD	1.2860	1.5855	0.0120	1.7851	1.3564	–	1.0116	0.1640
7.	AUD	1.2713	1.5674	0.0119	1.7647	1.3409	0.9886	–	0.162
8.	HKD	7.8403	9.6663	0.0732	10.8831	8.2695	6.0967	6.1672	–

ფორმალურად ეს ცხრილი $A = (a_{i,j})_{i,j=1}^{8}$ მატრიცის სახით შეგვიძლია ჩავწეროთ: თუ ყოველ ვალუტას მის ცხრილში მოცემულ ნომერს შევუსაბამებთ, $a_{i,j}$ აღნიშნავს j ვალუტის i ვალუტაზე გადაცვლის კოეფიციენტს (ჩვენს მაგალითში $a_{3,4} = 148.7210$ იაპონური იენის ბრიტანულ ფუნტზე გადაცვლის კურსს აღნიშნავს).



ნახ. 6.4: ვალუტის გადაცვლის გრაფი

ამ გრაფის ყოველი გზა ვალუტების გაცვლის მიმდევრობას აღნიშნავს. მაგალითად, 137452 ამერიკული დოლარის იაპონურ იენზე, შემდეგ ავსტრალიურ დოლარზე, მერე ბრიტანულ ფუნტზე, შვეიცარულ ფრანკსა და ბოლოს ევროზე გადაცვლის პროცესს აღნიშნავს. თუ დავიწყებთ ერთი ამერიკული დოლარით, ვამთავრებთ დოლარით, დავამთავრებთ $a_{3,1} \cdot a_{7,3} \cdot a_{4,7} \cdot a_{5,4} \cdot a_{2,5} = 107.14 \cdot 0.0119 \cdot 0.5667 \cdot 1.3161 \cdot 0,8555 = 0.8135$
თუ შემდეგ ევროებს ისევ დოლარში გადავახურდავებთ, მივიღებთ ციკლს და შევძლებთ იმის შეფასებას, მომგებიანი იყო თუ არა ჩატარებული ტრანსაქციები (ნახ. 6.4 წითლად ნაჩვენები წიბოები). ჩვენს შემთხვევაში მივიღებთ $0.8135 \cdot 1.2329 = 1.0029$, რაც იმას ნიშნავს, რომ ამ ოპერაციების ჩატარების შემდეგ მოგებულები დავრჩებით (მართალია პატარა პროცენტით, მაგრამ მაინც).

აქედან გამომდინარე, შეიძლება დაისვას შემდეგი ამოცანა: მოცემულ მიმართულ შეწონილ გრაფში აღმოაჩინე ისეთი ციკლი, რომ გავლილი წიბოების წონების წონების ნამრავლი ერთზე მეტ რიცხვს გვაძლევდეს.
აქამდე განხილულ ალგორითმებში ვეძებდით განვლილი წიბოების წონების *ჯამის* მინიმუმს (ან, ზოგ შემთხვევაში მაქსიმუმს).
როგორ შეიძლება წონების ჯამიდან ნამრავლზე გადასვლა? ამ შეკითხვაზე პასუხის გაცემა ლოგარითმების გამოყენებით შეიძლება:

$$x_1 \cdot x_2 \cdots x_n = 2^{\log(x_1 \cdot x_2 \cdots x_n)} = 2^{\log x_1 + \log x_2 + \cdots + \log x_n}$$

აქედან გამომდინარე, თუ გრაფის წიბოების წონად ავიღებთ აქამდე არსებული წონის (გადაცვლის კურსის) ლოგარითმს, მომგებიანად გადაცვლის კომბინაციის პოვნის ამოცანას დავიყვანთ ამ გრაფში მაქსიმალურად გრძელი გზის პოვნის ამოცანაზე (ან, ალტერნატიულად, ისეთი ციკლის პოვნის ამოცანაზე, რომლის წონაც ერთზე მეტია). აღნიშნულ ამოცანას ერთი ძალიან დიდი ნაკლი აქვს: იგი ე.წ. NP სრული ამოცანების კლასს განეკუთვნება, რომელთათვისაც პოლინომიური ალგორითმი ცნობილი არ არის (და არც ისაა ცნობილი, ამოხსნადია თუ არა ამ კლასის რომელიმე ამოცანა პოლინომიურ დროში). მაგრამ თუ წონებად ავიღებთ არა გადაცვლის კურსის, არამედ მისი შებრუნებული რიცხვის ლოგარითმს (ანუ $-\log a_{i,j}$), მაშინ მომგებიანი ციკლი უარყოფითი იქნება.
ამრიგად, ვალუტის მომგებიანად გადაცვლის ჯაჭვის პოვნის ამოცანა დაიყვანება გრაფში უარყოფითი ციკლების პოვნის ამოცანაზე.

თავი 7

არითმეტიკული ალგორითმები და მათი გამოყენება

რადგან ჩვენ ორობით სისტემაში მოქმედებას ვაპირებთ, აუცილებელია შესაბამისი ოპერაციების განსაზღვრაც. თუ ჩვენ ათობით არითმეტიკაში (შესაბამისად ალგებრაში) მიმატების, გამრავლების, გაყოფის ოპერაციები გვაქვს შემოღებული, ანალოგიური ოპერაციები უნდა შემოვიღოთ ორობით ალგებრაშიც, ანუ **ბულის ალგებრაში**, როგორც ამას მისი ფუძემდებლის, ინგლისელი მათემატიკოსის ჯორჯ ბულის (George Boole) პატივსაცემად უწოდებენ.

7.1 ბულის ალგებრის ელემენტები

ბულის ლოგიკა და, აქედან გამომდინარე, ბულის ალგებრა $\mathbb{B} = \{0, 1\}$ ორობით ანბანზეა განსაზღვრული. ზოგადად რომ ვთქვათ, ეს კლასიკური ლოგიკის მათემატიკურ ენაზე გადატანის ერთ-ერთი (ყველაზე გავრცელებული) მაგალითია. ლოგიკაში გვაქვს ჭეშმარიტი და მცდარი გამონათქვამები: ყოველი გამონათქვამი (მაგალითად, „$3+4=7$", „$12-3=1$", „ხვალ მზე ამოვა", „გუშინ წვიმდა" და ა.შ.) ***ან*** ჭეშმარიტია, ***ან*** მცდარი - სხვა რამ შეუძლებელია.
ბულის ძირითადი იდეა გამონათქვამების ***მათემატიკურ ცვლადებზე გადატანა*** იყო: ყოველი გამონათქვამი X ან ჭეშმარიტია (მაშინ $X = 1$), ან მცდარი ($X = 0$). ასევე შესაძლებელია გამონათქვამების კომბინირებაც, მაგალითად: „გუშინ მზე ამოვიდა ***და ამავდროულად*** წვიმდა", ან „$2 + 3 = 5$ და ამავდროულად $2 - 7 = 1$".
ამ მაგალითებში, თუ X =„გუშინ მზე ამოვიდა", Y =„წვიმდა", მაშინ სრული გამონათქვამი Z =„გუშინ მზე ამოვიდა ***და ამავდროულად*** წვიმდა" მათემატიკურად შემდეგნაირად ჩაიწერება: $Z = X\&Y$. საბოლოო ჯამში, თუ გუშინ მართლა ამოვიდა მზე ($X = 1$) და ამ დროს მართლაც წვიმდა ($Y = 1$), მაშინ $Z = X\&Y = 1$ ჭეშმარიტი იქნება.
მეორეს მხრივ, თუ X' =„$2 + 3 = 5$" (ჭეშმარიტია) და Y' =„$2 - 7 = 1$" (მცდარია), ცხადია, რომ $X' = 1$ და $Y' = 0$. აქედან გამომდინარე, $X'\&Y' = 0$ და გამონათქვამი Z =„$2 + 3 = 5$ და ამავდროულად $2 - 7 = 1$" მცდარია.
ანალოგიურად შეიძლება შემდეგი გამონათქვამების შედგენა: „ხვალ იწვიმებს ***ან*** ხვალ ქარი იქნება"; „$3+7 = 11$ ***ან*** $2 - 5 = -3$". ცხადია, რომ ასეთი გამონათქვამები ჭეშმარიტია, თუ ერთი მაინც ჭეშმარიტია. მათემატიკურად ეს შემდეგნაირად შეიძლება ჩამოყალიბდეს: X =„ხვალ იწვიმებს", Y =„ხვალ ქარი იქნება", $Z = X \vee Y$=„ხვალ იწვიმებს ***ან*** ხვალ ქარი იქნება"; X' =„$3 + 7 = 11$", Y' =„$2 - 5 = -3$", $Z' = X' \vee Y' = 1$: აქ ***ან*** ერთი უნდა შესრულებულიყო, ***ან*** მეორე.
მესამე მნიშვნელოვანი ოპერაცია ***უარყოფაა***: გამონათქვამის შებრუნებულის აღება.
მაგალითად, „ხვალ იწვიმებს"$\longrightarrow$„ხვალ *არ* იწვიმებს; „$3 + 7 = 13 \longrightarrow 3 + 7 \neq 13$" და ა.შ. X გამონათქვამის უარყოფა მათემატიკურად შემდეგნაირად ჩაიწერება: $\neg X$.
ბულის ალგებრის ეს სამი ოპერაცია ქართულ ენაზე შემდეგნაირად შეიძლება ჩამოვაყალიბოთ: გამონათქვამი $Z = X\&Y$ ჭეშმარიტია, თუ X ***და*** Y გამონათქვამი ორივე ჭეშმარიტია; $Z = X \vee Y$ ჭეშმარიტია, თუ X ***ან*** Y ჭეშმარიტია; $Z = \neg X$ ჭეშმარიტია, თუ X მცდარია.
ეს ყველაფერი მათემატიკერ ენაზე შემდეგნაირად ჩამოყალიბდება: ორი X, Y გამონათქვამის ***კონიუნქცია*** $X\&Y$ ორ ცვლადიან ფუნქციას $f : \mathbb{B}^2 \to \mathbb{B}$ ეწოდება, რომლის მნიშვნელობაა 1, თუ ორივე ცვლადის მნიშვნელობაა 1; ორი X', Y' გამონათქვამის ***დიზიუნქცია*** $X \vee Y$ ორ ცვლადიან ფუნქციას $g : \mathbb{B}^2 \to \mathbb{B}$ ეწოდება, რომლის მნიშვნელობაა 1, თუ ერთ-ერთი ცვლადის მნიშვნელობაა 1; Z გამონათქვამის ***უარყოფა*** $\neg Z$ ერთ ცვლადიან ფუნქციას $h : \mathbb{B} \to \mathbb{B}$ ეწოდება, რომლის მნიშვნელობაა 1, თუ Z ცვლადის მნიშვნელობაა 0.

ყოველივე ეს ცხრილის სახითაც შეიძლება გამოვსახოთ:

X	Y	$X\&Y$	$X \vee Y$	$\neg X$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

ამ ცხრილში მოცემულია აღსაწერი ფუნქციების მნიშვნელობები ცვლადების (ამ შემთხვევაში X და Y) ყველა შესაძლო კომბინაციისათვის.
შენიშვნა: კონიუნქცია და უარყოფა სხვადასხვანაირად აღიწერება ხოლმე. სიმარტივისთვის შეგვიძლია დავწეროთ: $X\&Y = X \cdot Y = XY$, $\neg X = \overline{X}$.

ბულის ალგებრაში უსასრულოდ ბევრი ფუნქციის მოყვანა შეიძლება, მაგრამ მთავარი ისაა, რომ ყველა ეს ფუნქცია ზემოთ მოყვანილი დიზიუნქციის, კონიუნქციისა და უარყოფის საშუალებით გამოისახება.
ფუნქციების მაგალითად შეგვიძლია მოვიყვანოთ:

$$\begin{aligned} f(x_1, x_2, x_3) &= \overline{x_1}x_2 \vee x_3, \\ g(x_1, x_2, x_3, x_4) &= (x_1 \vee x_2\overline{x_3} \vee \overline{x_4}, x_1x_2), \\ h(x_1, x_2) &= (x_1x_2, x_1 \vee x_2, x_1\overline{x_2} \vee \overline{x_1}x_2) \end{aligned}$$

ამ მაგალითში f ფუნქცია სამ ცვლადიანია (თითოეული ცვლადი $\mathbb{B}$ სიმრავლიდან) და ერთ ელემენტს გვაძლევს პასუხად (იგივე $\mathbb{B}$ სიმრავლიდან). ამ შემთხვევაში იტყვიან, რომ ეს ფუნქცია $\mathbb{B}^3$ სიმრავლეს ასახავს $\mathbb{B}$ სიმრავლეში: $f : \mathbb{B}^3 \to \mathbb{B}$.
g ფუნქცია 4 ცვლადს ასახავს ორ პარამეტრიან პასუხში, ესე იგი $g : \mathbb{B}^4 \to \mathbb{B}^2$, ხოლო $h\mathbb{B}^2 \to \mathbb{B}^3$.
ზოგადად, თუ რამიე ფუნქცია ϕ n ცვლადიანია, ხოლო ეს ცვლადები მნიშვნელობას რაიმე A სიმრავლიდან შეიძლება იღებდნენ და მისი პასიხი m პარამეტრიანია და ამ პასუხის ელემენტები C სიმრავლიდან შეიძლება იყოს, იტყვიან, რომ ეს ფუნქცია A^n სიმრავლეს (ანუ A სიმრავლის ელემენტებისაგან შემდგარ n სიგრძის სიტყვას - ვექტორს) ასახავს C^m სიმრავლეში (ანუ C სიმრავლის ელემენტებისაგან შემდგარ m სიგრძის სიტყვაში - ვექტორში).
მათემატიკურად ეს შემდეგნაირად ჩაიწერება: $\phi : A^n \to C^m$.
ამ თავში ჩვენ $f : \mathbb{B}^n \to \mathbb{B}^m$ ფუნქციებს განვიხილავთ. ასეთ ფუნქციებს დისკრეტულსაც, ანუ ყველგან წყვეტილს უწოდებენ. ანალოგიურად, ასეთ ფუნქციებზე შედგენილ მათემატიკას დისკრეტული მათემატიკა ეწოდება.

განვიხილოთ დისკრეტული ფუნქცია $f(x_1, x_2, x_3, x_4) = x_1\overline{x_2}x_3 \vee x_4 \vee \overline{x_2x_3} \vee x_1x_2x_3x_4$. ამ ფუნქციის გამოსათვლელად საჭიროა შემდეგი ბიჯები:

1. $z_1 = x_2x_3$;
2. $z_2 = x_1z_1 = x_1x_2x_3$;
3. $z_3 = z_2x_4 = x_1x_2x_3x_4$;
4. $z_4 = \overline{z_1} = \overline{x_2x_3}$;
5. $z_5 = \overline{x_2}$;
6. $z_6 = x_1z_5 = x_1\overline{x_2}$;
7. $z_7 = z_6x_3 = x_1\overline{x_2}x_3$;
8. $z_8 = z_7 \vee x_4 = x_1\overline{x_2}x_3 \vee x_4$;
9. $z_9 = z_8 \vee z_4 = x_1\overline{x_2}x_3 \vee x_4 \vee \overline{x_2x_3}$;
10. $z_{10} = z_9 \vee z_3 = x_1\overline{x_2}x_3 \vee x_4 \vee \overline{x_2x_3} \vee x_1x_2x_3x_4$.

ლოგიკურად ისმის ორი შეკითხვა: რამდენი ოპერაციის ჩატარება გვიხდება ამ გამოსახულების გამოსათვლელად? რამდენი ბიჯია (დროა) საჭირო ამ გამოსახულების გამოსათვლელად? ამ შეკითხვებზე პასუხის გაცემა შემდეგნაირად შეიძლება:

ოპერაციათა რაოდენობის დასათვლელად საკმარისია ლოგიკური ოპერაციების (კონიუნქცია, დიზიუნქცია, უარყოფა) დათვლა: $C(f(x_1,x_2,x_3,x_4)) = 10$.
რაც შეეხება დროს (ბიჯების რაოდენობას) $T(f(x_1,x_2,x_3,x_4))$, ზემოთ მოყვანილ გამოთვლის მეთოდში ეს ოპერაციათა რაოდენობას დაემთხვა, რადგან ჩვენ ყველა ოპერაციას რიგ-რიგობით ვატარებდით.
ამ მაგალითში გასათვალისწინებელია ის ფაქტი, რომ რამოდენიმე ოპერაცია ***ერთდროულად*** შეიძლება ჩატარდეს: მაგალითად, შესაძლებელია (x_1x_3), $\overline{x_2}$ და (x_2x_3) გამოსახულებების გამოთვლა, რადგან ისინი ერთმანეთზე დამოკიდებული არაა, განსხვავებით, მაგალითად, $y_1 = x_1x_2$ და $y_2 = x_1x_2x_3$ გამოსახულებელისაგან, რომელთა გამოთვლა ერთდროულად არ შეიძლება: ერთი მეორეზეა დამოკიდებული.

აქედან გამომდინარე, შეგვიძლია შემდეგი „პარალელური" ალგორითმის შემოთავაზება:

1. $z_1 = x_2x_3$ და ***ამავდროულად*** $z_2 = x_1x_4$ და ***ამავდროულად*** $z_5 = \overline{x_2}$ და ***ამავდროულად*** $z_6 = x_1x_3$;
2. $z_3 = z_1z_2 = x_1x_2x_3x_4$ და ***ამავდროულად*** $z_4 = \overline{z_1} = \overline{x_2x_3}$ და ***ამავდროულად*** $z_7 = z_5z_6 = x_1\overline{x_2}x_3$;
3. $z_8 = z_7 \vee x_4 = x_1\overline{x_2}x_3 \vee x_4$ და ***ამავდროულად*** $z_9 = z_4 \vee z_3 = \overline{x_2x_3} \vee x_1x_2x_3x_4$;
4. $z_{10} = z_8 \vee z_9 = x_1\overline{x_2}x_3 \vee x_4 \vee \overline{x_2x_3} \vee x_1x_2x_3x_4$.

აქ მნიშვნელოვანია ის ფაქტი, რომ გარკვეული ოპერაციები ***ერთდროულად*** სრულდება, რის ხარჯზეც ფუნქციის ***სიღრმე*** (ანუ გამოთვლის ბიჯების რაოდენობა) მცირდება.

აქედან გამომდინარე ვიღებთ შემდეგ განსაზღვრებას:

განმარტება 7.1: $f : \mathbb{B}^n \to \mathbb{B}^m$ ბულის ფუნქციის ოპერაციათა რაოდენობა $C(f(x_1,...,x_n)$ მასში შემავალი კონიუნქციის, დიზიუნქციისა და უარყოფების რაოდენობათა ჯამის ტოლია; იგივე ფუნქციის სიღრმე (რაც იგივე გამოთვლის დროა) $T(f(x_1,...,x_n))$ მისი რეალიზაციისათვის პარალელურად ჩატარებულ ოპერაციათა ბიჯების რაოდენობის ტოლია.

აღსანიშნავია, რომ თუ ფუნქცია მრავალგანზომილებიანია (ანუ მრავალცვლადიანია), როგორც, მაგალითად, $f(x_1,x_2) = (\overline{x_1} \vee \overline{x_2}, x_2)$, მისი ელემენტების რაოდენობის გამოსათვლელად უნდა დავითვალოთ ყველა პასუხისთვის (ამ შემთხვევაში $C(\overline{x_1} \vee \overline{x_2}) = 3$, $C(x_2) = 0$ და დავითვალოთ მათი ჯამი: $C(f(x_1,x_2)) = 3 + 0 = 3$, ხოლო სიღრმის დასათვლელად უნდა გამოვიანგარიშოთ თითოეულის სიღრმე და ავიღოთ მათი მაქსიმუმი (ფუნქიის ყოველი მნიშვნელობის გამოთვლა შეიძლება ერთდროულად): $T(\overline{x_1} \vee \overline{x_2}) = 2$, $T(x_2) = 0$, $T(\overline{x_1} \vee \overline{x_2}, x_2) = \max\{T(\overline{x_1} \vee \overline{x_2}), T(x_2)\} = \max\{2, 0\} = 2$.

სავარჯიშო 7.1: განიხილეთ ფუნქციები $f(x_1,x_2,x_3) = \overline{x_1}x_2 \vee x_3$, $g(x_1,x_2,x_3,x_4) = (x_1 \vee x_2\overline{x_3} \vee \overline{x_4}, x_1x_2)$ და $h(x_1,x_2) = (x_1x_2, x_1 \vee x_2, x_1\overline{x_2} \vee \overline{x_1}x_2)$. დაითვალეთ მათი ოპერაციათა რაოდენობა და სიღრმე.

იმ ამოცანებში, რომელთა განხილვას ჩვენ ვაპირებთ, მნიშვნელოვან როლს ორის მოდულით მიმატება ასრულებს. ეს განპირობებულია იმით, რომ კომპიუტერული სისტემები ორობით ანბანზეა აგებული.
ორობითი მიმატება (როგორც მას სხვანაირად უწოდებენ) განსაზღვრულია შემდეგნაირად:
ფუნქცია $f(x,y) = x \oplus y = 1$ მაშინ და მხოლოდ მაშინ, თუ მისი ცვლადებიდან ***ზუსტად ერთი*** ტოლია ერთის: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$.
როგორ შეიძლება ამ ფუნქციის კონიუნქციის, დიზიუნქციისა და უარყოფის მეშვეობით ჩაწერა? პირველ რიგში ქართულ ენაზე ჩამოვაყალიბოთ, თუ ცვლადების რა მნიშვნელობებისათვის ხდება ფუნქცია 1:
$f(x,y) = x \oplus y$ ფუნქცია ერთის ტოლი ხდება მაშინ და მხოლოდ მაშინ, თუ $x = 1$ და $y = 0$ ***ან*** $x = 0$ და $y = 1$. ლოგიკურად, თუ ცვლადი არის 1, მისი უარყოფა უნდა იყოს 0. ამიტომაც ვიღებთ შემდეგ გამონათქვამს:$f(x,y) = x \oplus y$ ფუნქცია ერთის ტოლი ხდება მაშინ და მხოლოდ მაშინ, თუ $x = 1$ და $\neg y = 1$ ***ან*** $\neg x = 1$ და $y = 1$. ეს შემდეგ ფუნქციას განაპირობებს: $f(x,y) = x \oplus y = \neg x \cdot y \vee \neg x \cdot y$. აქ „$x = 0$" (ანალოგიურად „$y = 0$") გამონათქვამების „$\neg x = 1$" („$\neg y = 1$") გამონათქვამებად შეცვლა იმიტომ დაგვჭირდა, რომ $x \cdot y$ გამოსახულება გამოსულიყო 1, თუ ***ზუსტად*** ერთი ცვლადია 1.

ზოგადად, თუ რაიმე უცნობი ფუნქცია მოცემულია ცხრილის სახით, მისი ფორმულებით ჩაწერა შემდეგნაირად შეიძლება:
მოცემულია ფუნქცია $f(x_1,x_2,...,x_n)$;

- გამოყავით ცვლადების ის კომბინაციები, რომლისთვისაც ფუნქცია ხდება 1 (ზედა შემთხვევაში ესაა $x = 0, y = 1$ და $x = 1, y = 0$);
- თითოეული ასეთი კომბინაციისათვის შეადგინეთ კონიუნქციებისაგან შემდგარი გამოსახულება. თუ $c_i = 0$, აიღეთ $\neg c_i$, წინააღმდეგ შემთხვევაში თვითონ c_i (ზედა შემთხვევაში ესაა $\neg x \cdot y$ და $x \cdot \neg y$);
- ეს გამოსახულებები შეაერთეთ დიზიუნქციებით (ზედა შემთხვევაში ვიღებთ $\neg x \cdot y \vee x \cdot \neg y$).

ასეთი სახით ჩაწერილ ფუნქციებს, სადაც კონიუნქციებით შეკრული ცვლადებით (ან მათი უარყოფებით) გამოსახულებები შეერთებულია დიზიუნქციებით, დიზიუნქციური ნორმალური ფორმა ეწოდება.

სავარჯიშო 7.2: ცრილით მოცემული ფუნქციები ჩაწერეთ დიზიუნქციური ნორმალური ფორმით:

x_0	x_1	x_2	f_1	f_2	f_3	f_4
0	0	0	1	0	1	1
0	0	1	0	1	0	0
0	1	0	0	1	1	0
0	1	1	1	0	1	1
1	0	0	1	1	0	0
1	0	1	1	1	1	1
1	1	0	0	0	1	0
1	1	1	0	0	0	0

აღსანიშნავია, რომ (ჩვეულებრივი ალგებრის მსგავსად) ჭეშმარიტია შემდეგი ტოლობები:

$$x(y \vee z) = x \cdot y \vee x \cdot z; \;\; (x \vee y)(x \vee z) = x \vee (yz); \;\; x \vee 0 = x; \;\; x \vee 1 = 1; \;\; x \cdot 0 = 0; \;\; x \cdot 1 = x.$$

სავარჯიშო 7.3: დაამტკიცეთ ზემოთ მოყვანილი ტოლობების ჭეშმარიტება (მოიყვანეთ თითოეული ფუნქციის ცხრილი და შეადარეთ მათი მნიშვნელობები).

სავარჯიშო 7.4: დაამტკიცეთ: $x \vee x \cdot y = x; \;\; \neg x \vee x \cdot y = \neg x \vee y$.

როგორც სიმრავლეთა თეორიაში, ასევე ბულის ლოგიკაშიც მნიშვნელოვანია ე.წ. დე მორგანის კანონები:

$$x \vee y = \overline{\overline{x} \cdot \overline{y}}; \;\; x \cdot y = \overline{\overline{x} \vee \overline{y}}.$$

სავარჯიშო 7.5: დაამტკიცეთ დე მორგანის კანონში მოყვანილი ფორმულები.

სავარჯიშო 7.6: რისი ტოლია $\neg(\neg x)$?

დე მორგანის კანონებზე დაყრდნობით დისიუნქციური ნორმალური ფორმის გადაყვანა შეიძლება ე.წ. კონიუნქციურ ნორმალურ ფორმაში - ისეთ გამოსახულებაში, რომელიც შედგება ცვლადების დიზიუნქციური გაერთიანებებით და ამ გამოსახულებათა კონიუნქციებით გაერთიანებებისაგან. კონიუნქციური ნორმალური ფორმით ჩაწერილი ფუნქციების მაგალითებია $(x_2 \vee \neg x_3)(x_1 \vee x_2 \vee x_3)(x_1 \vee x_2 \vee \neg x_3)$ და $(x_1 \vee x_3)(x_1 \vee x_2)(x_1 \vee \neg x_2 \vee x_3)$, მაგრამ არა $(x_2 \vee \neg x_3)(x_1 \vee x_2 \vee x_3)(x_1 \vee x_2 \vee \neg x_3) \vee x_1$.

თუ მოცემული გვაქვს რაიმე ფუნქცია, ზოგჯერ მისი ოპერაციებისა და ბიჯების რაოდენობის შემცირება შეიძლება ზემოთ მოყვანილი ტოლობების მეშვეობით: $(x_1 \vee x_2\overline{x_3} \vee \overline{x_4}x_1x_2) = x_1(1 \vee \overline{x_4}x_2) \vee x_2\overline{x_3} = x_1 \vee x_2\overline{x_3}$.
იგივე ფუნქციის კონიუნქციური ნორმალური ფორმით ჩაწერა შემდეგნაირად შეიძლება:
$x_1 \vee x_2\overline{x_3} = \neg(\overline{x_1} \cdot \overline{(x_2\overline{x_3})}) = \neg(\overline{x_1} \cdot (\overline{x_2} \vee x_3))$.

სავარჯიშო 7.7: შემდეგი ფუნქციები ჩაწერეთ დიზიუნქციური ნორმალური ფორმით:

$$\begin{aligned} f(x_1, x_2, x_3) &= (x_2 \vee \neg x_3)(x_1 \vee x_2 \vee x_3)(x_1 \vee x_2 \vee \neg x_3); \\ g(x_1, x_2, x_3) &= (x_1 \vee x_3)(x_1 \vee x_2)(x_1 \vee \neg x_2 \vee x_3); \\ h(x_1, x_2, x_3) &= (x_1 \vee \neg x_2)(x_1 \vee x_2)(x_1 \vee x_2 \vee x_3). \end{aligned}$$

7.2 n ბიტიანი რიცხვების მიმატება

მიმატების ოპერაცია იმდენად ხშირია ჩვენს ყოველდღიურ ცხოვრებაში, რომ ბევრ ადამიანს, ალბათ, არც კი მოსვლია თავში აზრად ის ფაქტი, რომ ეს პროცესი არც თუ ისე მარტივია. მაგალითად, ყველა სწრაფად გამოგვითვლის $5+3=8$, მაგრამ $3434164136861+3289747301047=6723911437908$ არც თუ ისე მცირე დროსა და ყურადღებას მოითხოვს. ზოგადად, რაც უფრო გრძელია შესაკრები რიცხვები, მით უფრო დიდ დროს ვანდომებთ გამოთვლას.
მიუხედავად იმისა, რომ შეკრების ყველაზე მარტივი ალგორითმი - ქვეშ მიწერით მიმატება - საყოველთაოდ ცნობილია, ჩვენ მაინც შევეცდებით მის განხილვასა და გაანალიზებას და ამას როგორც ათობით, ასევე ორობითი რიცხვების მაგალითზე გავაკეთებთ.

ქვეშ მიწერით მიმატების მეთოდი

საყოველთაოდ ცნობილი მეთოდის გარჩევა მარტივი მაგალითით დავიწყოთ:

$$+\begin{array}{r}427\\613\\\hline 0\end{array}\ (1)\qquad +\begin{array}{r}427\\613\\\hline 40\end{array}\ (0)\qquad +\begin{array}{r}427\\613\\\hline 040\end{array}\ (1)\qquad +\begin{array}{r}427\\613\\\hline 1040\end{array}$$

ზოგადად, თუ მოცემულია ორი n ციფრიანი რიცხვი $a_{n-1}...a_0$ და $b_{n-1}...b_0$, მისი ჯამი $d_n...d_0$ შემდეგი ალგორითმით შეიძლება გამოვიანგარიშოთ:

```
c_0 = 0;
for( i = 0, i < n, i++)
  {
    d_i = a_i + b_i + c_i mod 10;
    if( a_i + b_i + c_i > 9 )
      c_{i+1} = 1;
      else c_{i+1} = 0;
  }
d_n = c_n;
return(d_n, ..., d_0)
```

იმის დასამტკიცებლად, რომ მოყვანილი ალგორითმი მართლაც სწორ შედეგს მოგვცემს, საჭიროა შემდეგი მათემატიკური ფორმულა: $(x_n x_{n-1}...x_0) = 10^n \cdot x_n + 10^{n-1} \cdot x_{n-1} + ... + 10^0 \cdot x_0$.

სავარჯიშო 7.8: დაამტკიცეთ ტოლობა $(x_n x_{n-1}...x_0) = 10^n \cdot x_n + 10^{n-1} \cdot x_{n-1} + ... + 10^0 \cdot x_0$.

სავარჯიშო 7.9: წინა სავარჯიშოს შედეგის გამოყენებით დაამტკიცეთ ზემოთ მოყვანილი ალგორითმის სისწორე.
იგივე მეთოდით ორობითი როცხვების შეკრებაც შეგვიძლია:
მოცემულია ორი n ბიტიანი ორობითი რიცხვი $a=(a_{n-1}...a_0)_2$ და $b=(b_{n-1}...b_0)_2$. გამოიანგარიშეთ მისი ჯამი $(d_n...d_0)_2$:

$$+\begin{array}{ccccc} & a_{n-1} & ... & a_1 & a_0\\ & b_{n-1} & ... & b_1 & b_0\\\hline d_n & d_{n-1} & ... & d_1 & d_0\end{array}$$

```
c_0 = 0;
for( i = 0, i < n, i++)
  {
    d_i = a_i + b_i + c_i mod 2;
    if( a_i + b_i + c_i ≥ 2 )
      c_{i+1} = 1;
      else c_{i+1} = 0;
  }
d_n = c_n;
return(d_n, ..., d_0)
```

სავარჯიშო 7.10: დაამტკიცეთ ტოლობა $(x_n x_{n-1} ... x_0)_2 = 2^n \cdot x_n + 2^{n-1} \cdot x_{n-1} + ... + 2^0 \cdot x_0$.

სავარჯიშო 7.11: წინა სავარჯიშოს შედეგის გამოყენებით დაამტკიცეთ ზემოთ მოყვანილი ალგორითმის სისწორე.

აღსანიშნავია, რომ $c_{i+1} = 1$ მაშინ და მხოლოდ მაშინ, თუ a_i, b_i და c_i ცვლადებს შორის ორი ან სამი ერთის ტოლია. ამის განსაზღვრა შემდეგნაირად შეიძლება: თუ $a_i = b_i = 1$, მაშინ პირობა სრულდება. თუ ამ ორი ცვლადიდან ზუსტად ერთია ერთის ტოლი, მაშინ ამავდროულად მესამე ცვლადიც (ანუ c_i) უნდა იყოს 1. იმის დადგენა, არის თუ არა ორი ცვლადიდან ზუსტად ერთი ერთიანის ტოლი, შეიძლება ორის მოდულით მიმატებით: $a_i \oplus b_i$. აქედან გამომდინარე, იმის დადგენა, გვხვდება თუ არა ორი ან სამი ერთიანი სამ ცვლადში, შემდეგი ფორმულით შეიძლება: $c_{i+1} = a_i b_i \vee (a_i \oplus b_i) c_i$ (აღსანიშნავია, რომ $a_i b_i = 1$ მაშინ და მხოლოდ მაშინ, თუ ორივე ცვლადი არის 1).

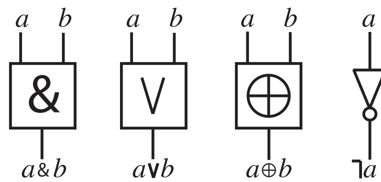
აქედან გამომდინარე d_i და c_i $(i = 1, ..., n-1)$ ცვლადების გამოსათვლელად გვაქვს შემდეგი ფორმულები:

$d_i = a_i \oplus b_i \oplus c_i$,
$c_{i+1} = a_i b_i \vee (a_i \oplus b_i) c_i$,
$d_n = c_n$.

მაგალითი:

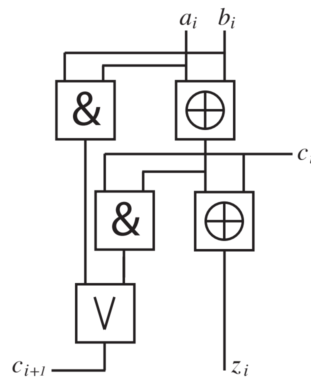
	8	7	6	5	4	3	2	1	0
a	0	1	1	0	0	1	0	0	1
b	0	1	1	1	1	0	0	1	0
d	1	1	0	1	1	1	0	1	1
c	1	1	1	0	0	0	0	0	0

ზემოთ მოყვანილი ბულის ალგებრის ფორმულები გრაფიკულადაც შეიძლება გამოვსახოთ:



ნახ. 7.1: ლოგიკური ოპერაციების გრაფიკული გამოსახვა

კონიუნქიის, დიზიუნქციისა და უარყოფის ოპერაციებს ბულის ალგებრის ელემენტარულ ოპერაციებსაც უწოდებენ.
აქედან გამომდინარე, შეგვიძლია შევადგინოთ d_i და c_i ცვლადების გამოსათვლელი სქემა:

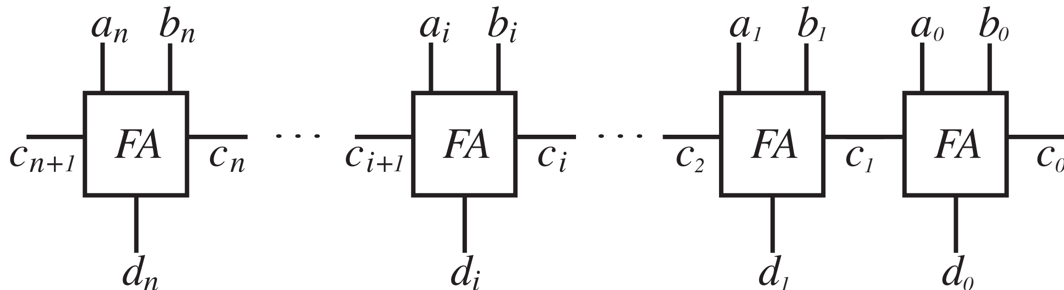


ნახ. 7.2: d_i და c_{i+1} ცვლადების გამოსათვლელი სქემა

აღსანიშნავია, რომ $\oplus$ ოპერაცია იმდენად ხშირად გამოიყენება, რომ მისი სქემა ერთი სიმბოლოთია აღნიშნული, თუმცა იგი რამოდენიმე ოპერაციის შესრულებას მოითხოვს.

სავარჯიშო 7.12: გამოიანგარიშეთ, რისი ტოლია $T(\oplus)$ და $C(\oplus)$.

თუ ჩვენ ამ სქემას აღვნიშნავთ როგორც FA (ინგლისური Full Adder, ანუ სრული შემკრები), მაშინ ორი $n+1$ ბიტიანი რიცხვის შეკრებისათვის საჭირო სქემა (რომელსაც ვუწოდებთ CRA_{n+1}) შემდეგნაირი იქნება:



ნახ. 7.3: ორი $n+1$ ბიტიანი რიცხვის შეკრებისათვის საჭირო სქემა CRA_{n+1}

სავარჯიშო 7.13: გამოიანგარიშეთ, რისი ტოლია $T(FA)$ (ანუ იმ ბიჯების რაოდენობა, რაც საჭიროა FA სქემის ყველა შედეგის გამოსაანგარიშებლად) და $C(FA)$ (ანუ FA სქემაში არსებული ელემენტების, იგივე ოპერაციების, რაოდენობა), თუ $T(\&) = T(\vee) = T(\neg) = 1$, და $C(\&) = C(\vee) = C(\neg) = 1$. აქვე გამოიყენეთ წინა სავარჯიშოში გამოთვლილი $T(\oplus)$ და $C(\oplus)$.

შენიშვნა: ხშირად იღებენ $T(\neg) = 0$ და $C(\neg) = 0$, ანუ სქემებში უარყოფის ელემენტებს უგულებელყოფენ იმის გამო, რომ მათი რეალიზაცია სხვა ელემენტების რეალიზაციასთან შედარებით საკმაოდ მცირეა და, ამავე დროს, უარყოფებს ხშირად იყენებენ დამხმარე ელემენტებად (სხვადასხვა ტექნიკური მიზეზებით ორ ერთმანეთზე მიყოლებულ უარყოფას სვამენ ხოლმე). ამას გარდა, ტექნიკურად შესაძლებელია ელემენტების სქემის ისეთი რეალიზაცია, რომ $T(\neg(ab)) = T(ab)$, $T(\neg(a \vee b)) = T(a \vee b)$, $T(\neg ab) = T(ab)$, $T(\neg a \vee b) = T(\neg a \vee b)$.

სავარჯიშო 7.14: გამოიანგარიშეთ, რისი ტოლია $T(\oplus)$, $C(\oplus)$ და, აქედან გამომდინარე, $T(FA)$ იმის გათვალისწინებით, რომ $T(\neg) = C(\neg) = 0$.

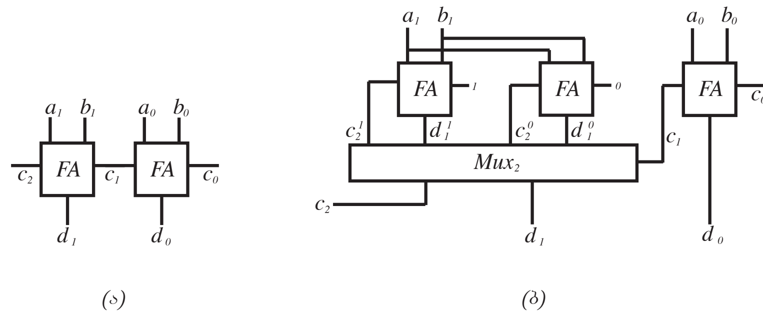
ადვილი დასანახია, რომ d_1 ცვლადი არ გამოითვლება, სანამ არ იქნება გამოთვლილი c_1 და, ზოგადად, d_i ცვლადის გამოთვლა არ შეიძლება, სანამ არ იქნება გამოთვლილი c_{i-1}. აქედან გამომდინარე, $T(CRA_n) = 4n$, $C(CRA_n) = 9n$ (აქ და შემდგომში დავუშვებთ, რომ $T(\neg) = C(\neg) = 0$).

სავარჯიშო 7.15: დაამტკიცეთ $T(CRA_n) = 4n$ და $C(CRA_n) = 9n$ ტოლობები.

სავარჯიშო 7.16: დახაზეთ CRA_1, CRA_2, CRA_3 და CRA_4 სქემები.

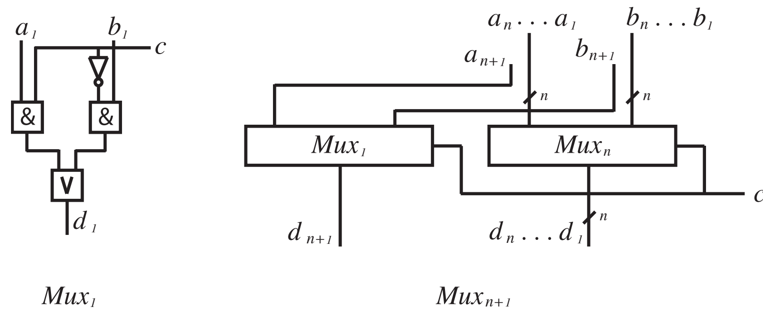
ბუნებრივია შემდეგი შეკითხვა: შესაძლებელია თუ არა ბიჯების რაოდენობისა და ელემენტების რიცხვის შემცირება?
თუ დავაკვირდებით ნახ. 7.4 *(ა)*-ში პირველ ორ FA ელემენტს, დავინახავთ შემდეგ მნიშვნელოვან ფაქტს: მარცხენა FA ელემენტი, რომელიც d_1 და c_2 ცვლადებს ითვლის, „ელოდება" c_1 ცვლადის გამოთვლას.
იმის გამო, რომ მას შეიძლება მიეწოდოს მხოლოდ $c_1 = 0$ ან $c_1 = 1$, ჩვენ შეგვიძლია ***ერთდროულად*** გამოვითვალოთ d_1 და c_2 იმ შემთხვევისათვის, როდესაც $c_1 = 0$ (d_1^0, c_2^0) და იმ შემთხვევისათვის, როდესაც $c_1 = 1$ (d_1^1, c_2^1).შემდეგ, როდესაც c_1 გამოთვლილი იქნება, შეიძლება ამ ორი საშუალედო შედეგიდან ერთ-ერთის არჩევა (ნახ. 7.4 *(ბ)*).
ამ ნახაზში გვხვდება ახალი ელემენტი Mux_2, რომლის მუშაობის შედეგები შემდეგი ცხრილით შეიძლება გამოისახოს:

$$d_1 = \begin{cases} d_1^0, & \text{თუ } c_1 = 0, \\ d_1^1, & \text{თუ } c_1 = 1 \end{cases} \qquad c_2 = \begin{cases} c_2^0, & \text{თუ } c_1 = 0, \\ c_2^1, & \text{თუ } c_1 = 1. \end{cases}$$



ნახ. 7.4:   ორი n ბიტიანი რიცხვის მიმატების პარალელური სქემა

ზოგადად, Mux_n შემდეგნაირად შეიძლება აღიწეროს (ნახ. 7.5) :

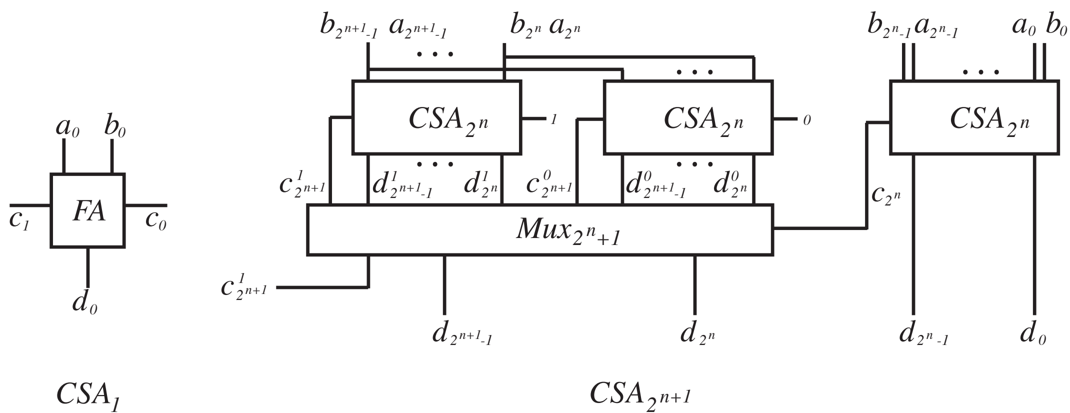


ნახ. 7.5: Mux_{n+1} – ორი $n+1$ ბიტიანი რიცხვის ამორჩევის სქემა

$$d_i = \begin{cases} b_i, & \text{თუ } c = 0, \\ a_i, & \text{თუ } c = 1. \end{cases} \qquad i = \overline{1; n+1}.$$

ნახ. 7.4 -ში მოყვანილ სქემას უწოდებენ CSA_2 (Carry Save Adder – Carry bit დახსომებულ ბიტს ეწოდება, Select - შერჩევა, Adder -შემკრები). იგი ორ 2 ბიტიან რიცხვს $a = (a_1, a_0)$ და $b = (b_1, b_0)$ შეკრებს და 3 ბიტიან რიცხვს $z = (c_2, d_1, d_0)$ მოგვცემს პასუხად.

ზოგადად, $CSA_{2^{n+1}}$, რომელიც ორ 2^{n+1} ბიტიან რიცხვს $A = (a_{2^{n+1}-1}, ..., a_0)$ და $B = (b_{2^{n+1}-1}, ..., b_0)$ შეკრებს და $2^{n+1} + 1$ ბიტიან რიცხვს $D = (c_{2^{n+1}}, d_{2^{n+1}-1}, ..., d_0)$ მოგვცემს პასუხად, შემდეგნაირად აღიწერება (ნახ. 7.6):



ნახ. 7.6: $CSA_{2^{n+1}}$ – ორი 2^{n+1} ბიტიანი რიცხვის შეკრების პარალელური სქემა

CSA_1, ანუ ორი ერთბიტიანი რიცხვის შემკრები არის ზემოთ განხილული სქემა FA.
ძირითადი იდეა „დაყავი და იბატონე" პარადიგმაზეა აგებული: მონაცემები ორ ნაწილად იყოფა —

$$A_1 = (a_{2^{n+1}-1}...a_{2^n}) \quad A_0 = (a_{2^n-1}...a_0)$$
$$B_1 = (b_{2^{n+1}-1}...b_{2^n}) \quad B_0 = (b_{2^n-1}...b_0)$$

შემდეგ გამოითვლება $D_0 = A_0 + B_0$ და ***ამავდროულად*** $D_1^0 = A_1 + B_1 + 0$ და $D_1^1 = A_1 + B_1 + 1$. ამის შემდეგ, c_{2^n} სიგნალის მეშვეობით, ამოირჩევა

$$D_1 = \begin{cases} D_1^0, & \text{თუ } c_{2^n} = 0, \\ D_1^1, & \text{თუ } c_{2^n} = 1 \end{cases} \qquad c_{2^{n+1}} = \begin{cases} c^0_{2^{n+1}}, & \text{თუ } c_{2^n} = 0, \\ c^1_{2^{n+1}}, & \text{თუ } c_{2^n} = 1 \end{cases}$$

აქ $D_0 = (d_{2^n-1}, ..., d_0)$ და $D_1 = (d_{2^{n+1}-1}, ..., d_{2^n})$.

ადვილი დასამტკიცებელია ამ სქემის სისწორე მათემატიკურ ინდუქციაზე დაყრდნობით:

- ინდუქციის შემოწმება: თუ $n = 0$, ცხადია, რომ $CSA_1 = FA$ და იგი ორ ერთ ბიტიან რიცხვს სწორად შეკრებს;
- ინდუქციის დაშვება: დავუშვათ, CSA_{2^n} სწორად შეკრებს ორ 2^n ბიტიან რიცხვს;
- ინდუქციის ბიჯი: დავამტკიცოთ, რომ $CSA_{2^{n+1}}$ სწორად შეკრებს ორ 2^{n+1} ბიტიან რიცხვს.

სავარჯიშო 7.17: დაამტკიცეთ, რომ თუ CSA_{2^n} სწორად შეკრებს ორ 2^n ბიტიან რიცხვს, მაშინ $CSA_{2^{n+1}}$ სწორად შეკრებს ორ 2^{n+1} ბიტიან რიცხვს.

რაც შეეხება ამ ალგორითმის ბიჯების რაოდენობას $T(CSA_{2^{n+1}})$, მისი გამოთვლა შემდეგნაირად შეიძლება:
უპირველესად ყოვლისა, უნდა გამოვითვალოთ ცვლადები D_0, D_1^0 და D_1^1, რაც ***ერთდროულად*** შეიძლება მოხდეს $T(CSA_{2^n})$ ბიჯში. ამის შემდეგ უნდა ავირჩიოთ D_1^0 და D_1^1 ცვლადებიდან ერთ-ერთი Mux_{2^n+1} სქემის საშუალებით, რაც $T(Mux_{2^n+1}) = 2$ ბიჯშია შესაძლებელი.
აქედან გამომდინარე, $T(CSA_{2^{n+1}}) = T(CSA_{2^n}) + T(Mux_{2^n+1}) = T(CSA_{2^n}) + 2$. ამ რეკურსიული ფორმულის გახსნის შემდეგ მივიღებთ:

$$T(CSA_{2^{n+1}}) = O(\log n).$$

სავარჯიშო 7.18: დაამტკიცეთ ტოლობა $T(Mux_{2^n+1}) = 2$ (გამოიყენეთ ნახ. 7.5-ში მოყვანილი რეკურსიული სქემა).

$C(CSA_{2^{n+1}})$ ოპერაციათა რაოდენობის გამოსათვლელად გამოვიყენოთ ფორმულა:

$$C(CSA_{2^{n+1}}) = 3 \cdot C(CSA_{2^n}) + C(Mux_{2^n+1}).$$

სავარჯიშო 7.19: დაამტკიცეთ, რომ $C(CSA_{2^{n+1}})$ მართლაც ამ რეკურსიული ფორმულით გამოითვლება და გამოითვალეთ მისი მნიშვნელობა.

როგორც ვხედავთ, პარალელური ალგორითმებით შეიძლება შეკრების ამოცანის სწრაფად გადაჭრა: ორი n ბიტიანი რიცხვისათვის არა $O(n)$, არამედ $O(\log n)$ ბიჯია საჭირო, სამაგიეროდ იზრდება ელემენტების რაოდენობა. ეს გასაკვირი არ არის: მეტ ოპერაციას ვატარებთ, ოღონდ ერთდროულად და ამის ხარჯზე ვიგებთ დროს.
აქვე უნდა აღინიშნოს, რომ არსებობს ორი n ბიტიანი რიცხვის მიმატების პარალელური ალგორითმი, რომლის დროის ზედა ზღვარია $O(\log n)$ და ელემენტების რაოდენობის ზედა ზღვარია $O(n)$ (სხვა სიტყვებით რომ ვთქვათ, შესაძლებელია ლოგარითმულ დროში გამოთვლა ისე, რომ ელემენტების რაოდენობა ძალიან არ გაიზარდოს), მაგრამ მათი განხილვა ჩვენი კურსის პროგრამას ცდება.

7.3 n ბიტიანი რიცხვების გამოკლება

წინა პარაგრაფში განხილული ალგორითმებით ფიქსირებული $n \in \mathbb{N}$ ბიტიანი სისტემების აგება შეიძლება. როდესაც აწყობილია სისტემა ფიქსირებული n ბიტიანი ორობითი რიცხვების დასამუშავებლად, მაქსიმალური რიცხვი, რაც შეიძლება წარმოვადგინოთ, იქნება $2^n - 1$: $(11...1)_2$. მასზე ერთით მეტი რიცხვი უკვე $n+1$ ბიტიანი იქნება: $(100...0)$, სადაც მარჯვენა n ბიტი ნულის ტოლია, ანუ თუ ჩვენ მხოლოდ n ბიტიან რიცხვებს განვიხილავთ და უფრო მაღალ ბიტებს უბრალოდ ვაგდებთ, ნებისმიერ არითმეტიკულ ოპერაციას 2^n მოდულით არითმეტიკაში ვატარებთ, რაც იმას ნიშნავს, რომ ნებისმიერი $0 \leq x < 2^n$ რიცხვისათვის შეგვიძლია გამოვიანგარიშოთ ისეთი შესაბამისი y, რომ $x + y = 0 \; mod \; 2^n$, ან, სხვა სიტყვებით რომ ვთქვათ, x რიცხვის შებრუნებული (უარყოფითი) მოდულით 2^n.
ბუნებრივია შეკითხვა: როგორ შეიძლება გამოვიანგარიშოთ მოცემული x რიცხვის შებრუნებული y? თუ განვიხილავთ რიცხვს $0 \; mod \; 2^n = 2^n = (11...1)_2 + 1_2 \; mod \; 2^n$, შეიძლება დავასკვნათ, რომ x რიცხვის შებრუნებულის გამოსათვლელად უნდა გამოვიანგარიშოთ ისეთი z რიცხვი, რომ $x + z = (11...1)_2$ და შემდეგ $y = z + 1$.

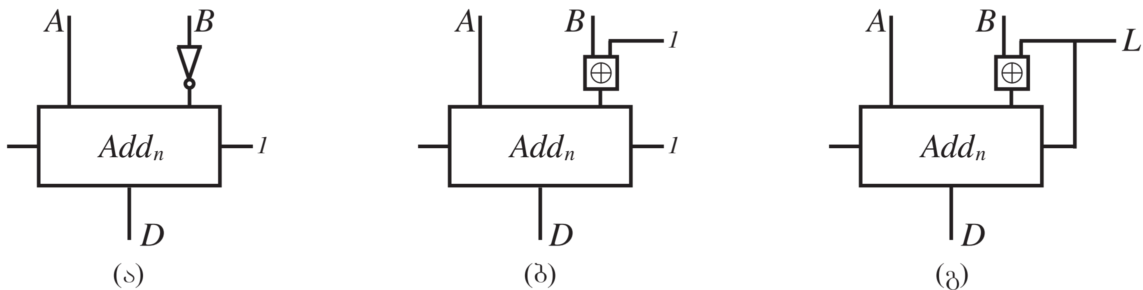
სავარჯიშო 7.20: დაამტკიცეთ, რომ თუ მოცემული n ბიტიანი x რიცხვისთვის მოვძებნით ისეთ z რიცხვს, რომ $x + z = (11...1)_2$, მაშინ x რიცხვის შებრუნებული (მოდულით 2^n) იქნება $y = z + 1$.

ცხადია, თუ ავიღებთ $x = (x_{n-1}x_{n-2}...x_0)_2$, მაშინ $z = \overline{x} = (\overline{x}_{n-1}\overline{x}_{n-2}...\overline{x}_0)_2$.

სავარჯიშო 7.21: დაამტკიცეთ, რომ მოცემული $x = (x_{n-1}x_{n-2}...x_0)_2$ და $z = (\overline{x}_{n-1}\overline{x}_{n-2}...\overline{x}_0)_2$ რიცხვებისათვის ჭეშმარიტია ტოლობა $x + z = 0 \; mod \; 2^n$.

აქედან გამომდინარე, $x = (x_{n-1}x_{n-2}...x_0)_2$ რიცხვის შებრუნებულია $y = \overline{x}+1 \; mod \; 2^n = (\overline{x}_{n-1}\overline{x}_{n-2}...\overline{x}_0)_2+1 \; mod \; 2^n$.

ყოველივე ზემოთ თქმულიდან შეიძლება დავასკვნათ, რომ მოცემული $a = (a_{n-1}a_{n-2}...a_0)_2$ და $b = (b_{n-1}b_{n-2}...b_0)_2$ რიცხვის სხვაობის გამოსათვლელად უნდა გამოვითვალოთ შემდეგი ჯამი: $d = a - b \; mod \; 2^n = a + \overline{b} + 1 \; mod \; 2^n$. ეს კი ნახ. 7.7(ა)-ში ნაჩვენებ სქემას მოგვცემს.



ნახ. 7.7: ორი n ბიტიანი რიცხვის გამოკლების სქემა (ა), (ბ) და მიმატება-გამოკლების კომბინირებული სქემა (გ)

აქ Add_n ორი n ბიტიანი რიცხვის მიმატების სქემაა, რომლის კონკრეტული რეალიზაცია ამ შემთხვევაში მნიშვნელოვანი არაა.

სავარჯიშო 7.22: დაამტკიცეთ, რომ ნახ. 7.7(ა)-ში ნაჩვენებ სქემა მართლაც A და B რიცხვების სხვაობას გამოითვლის.

რადგან $\neg x = x \oplus 1$, 7.7(ა) და 7.7(ბ) ნახაზებში ნაჩვენები სქემები ერთსა და იგივე შედეგს იძლევა. აქედან გამომდინარე, შესაძლებელია 7.7(გ) ნახაზში ნაჩვენები სქემით შეკრებისა და გამოკლების ერთიანი სქემის შექმნა: თუ მაკონტროლებელი სიგნალი $L = 1$, შესრულდება გამოკლება, ხოლო თუ $L = 0$ — მიმატება.

სავარჯიშო 7.23: დაამტკიცეთ, რომ $\neg x = x \oplus 1$ და $x = x \oplus 0$.

სავარჯიშო 7.24: დაამტკიცეთ, რომ თუ 7.7(გ) ნახაზში ნაჩვენებ სქემაში მაკონტროლებელი სიგნალი $L = 1$, შესრულდება გამოკლება, ხოლო თუ $L = 0$ — მიმატება.

აღსანიშნავია, რომ რეალურ სისტემებში შედეგი $D = (d_n, d_{n-1}...d_0)_2$ (და საერთოდ რიცხვები) $n+1$ ბიტიანი სიტყვებია, რომლებშიც უფროსი ბიტი d_n ნიშანს გვიჩვენებს: თუ $d_n = 1$, D რიცხვი უარყოფითია, წინააღმდეგ შემთხვევაში კი დადებითი. მაგრამ ჩვენ 2^n მოდულით არითმეტიკული ოპერაციებით შემოვიფარგლებით (ნიშნის გარეშე), რითიც უფრო ადვილია ძირითადი პრინციპების გაგება და შემდგომ სხვადასხვა პრაქტიკული რეალიზაციის ათვისება.

7.4 n ბიტიანი რიცხვების გამრავლება

7.4.1 ქვეშ მიწერით გამრავლება

„ქვეშ მიწერით გამრავლება" პირველი ალგორითმია, რომელსაც სკოლაში ვსწავლობთ:

$$\times \begin{array}{r} 427 \\ 613 \\ \hline 1281 \end{array} \quad \times \begin{array}{r} 427 \\ 613 \\ \hline 1281 \\ 427 \end{array} \quad \times \begin{array}{r} 427 \\ 613 \\ \hline 1281 \\ 427 \\ 2562 \end{array} \quad \times \begin{array}{r} 427 \\ 613 \\ \hline 1281 \\ 427 \\ 2562 \\ \hline 261751 \end{array}$$

$A = (a_{n-1}...a_0)$ და $B = (b_{n-1}...b_0)$ რიცხვების გადასამრავლებლად გამოიანგარიშე $C_k = 10^k \cdot A \cdot b_k$ და მათი ჯამი $C = C_0 + C_1 + \cdots + C_{n-1}$. აღსანიშნავია, რომ $A \cdot b_k$ რიცხვის გამოსათვლელად ცალკე ალგორითმია საჭირო, ხოლო $10^k x$ მოცემული x რიცხვის k პოზიციით მარცხნივ „ჩაჩოჩებას" ნიშნავს (ან მარჯვნივ შესაბამისი რაოდენობის ნულების მიწერას).

სავარჯიშო 7.25: დაამტკიცეთ, რომ ზემოთ მოყვანილი ქვეშ მიწერით გამრავლების მეთოდი მართლაც სწორ პასუხს იძლევა.
მინიშნება: $B = (b_{n-1}...b_0)$ რიცხვი წარმოადგინეთ შემდეგი ჯამის სახით: $B = 10^{n-1} \cdot b_{n-1} + \cdots + 10^0 \cdot b_0$.

ანალოგიურად შეგვიძლია გამოვიანგარიშოთ ორობითში წარმოდგენილი რიცხვების ნამრავლიც:

$$\times \begin{array}{r} 10110 \\ 10011 \\ \hline 10110 \\ 10110 \\ 00000 \\ 00000 \\ 10110 \\ \hline 110100010 \end{array}$$

ცხადია, რომ $A = (a_{n-1}...a_0)$ და $B = (b_{n-1}...b_0)$ ორობითი რიცხვის გამრავლების შემთხვევაში შემდეგნაირად უნდა მოვიქცეთ: გამოვიანგარიშოთ $C_k = 2^k \cdot A \cdot b_k = 2^k \cdot A \,\&\, b_k$ და მათი ჯამი $C = C_0 + C_1 + \cdots + C_{n-1}$.

სავარჯიშო 7.26: ათობითი რიცხვების ალგორითმის ანალოგიურად დაამტკიცეთ ამ მეთოდის სისწორე.

სავარჯიშო 7.27: დაამტკიცეთ, რომ ქვეშ მიწერით გამრავლების მეთოდის ოპერაციათა რაოდენობის ზედა ზღვარია $O(n^2)$. რა არის მისი ბიჯების რაოდენობის ზედა ზღვარი? შეიძლება თუ არა ამ მეთოდის პარალელიზაცია?

სავარჯიშო 7.28: განიხილეთ ათობითში ჩაწერილი n ბიტიანი რიცხვების ქვეშ მიწერით გამრავლების ალგორითმი $MultDec_n$ და ანალოგიური ალგორითმი $MultBin_n$, რომელიც ორობითში ჩაწერილ n ბიტიან რიცხვებს ამრავლებს. რა განსხვავებაა $C(MultDec_n)$ და $C(MultBin_n)$ ზედა ზღვრებს შორის? $T(MultDec_n)$ და $T(MultBin_n)$ ზედა ზღვრებს შორის? პასუხი დაამტკიცეთ.

7.4.2 გამრავლების პარალელური მეთოდი: ვოლესის ხე (Wallace Tree)

1964 წელს ავსტრალიელმა მეცნიერმა კრის ვოლესმა (Chris Wallace) გამრავლების პარალელიზაციის იდეა წამოაყენა, რომელსაც ჩვენს მაგალითზე განვიხილავთ.

C_i ცვლადები გამოვიანგარიშოთ როგორც ქვეშ მიწერით მეთოდში:

$$
\begin{array}{rr}
\times & 10110 \\
 & 10011 \\ \hline
C_0 & 10110 \\
C_1 & 10110 \\
C_2 & 00000 \\
C_3 & 00000 \\
C_4 & 10110 \\ \hline
C & 110100010 \\
\text{დახსომებული} & 001111100
\end{array}
\qquad
\begin{array}{rcccccccccc}
\times & & & & & & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & & & b_4 & b_3 & b_2 & b_1 & b_0 \\ \hline
C_0 & & & & & & c_{0,4} & c_{0,3} & c_{0,2} & c_{0,1} & c_{0,0} \\
C_1 & & & & & c_{1,4} & c_{1,3} & c_{1,2} & c_{1,1} & c_{1,0} & \\
C_2 & & & & c_{2,4} & c_{2,3} & c_{2,2} & c_{2,1} & c_{2,0} & & \\
C_3 & & & c_{3,4} & c_{3,3} & c_{3,2} & c_{3,1} & c_{3,0} & & & \\
C_4 & & c_{4,4} & c_{4,3} & c_{4,2} & c_{4,1} & c_{4,0} & & & & \\ \hline
C & c_9 & c_8 & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

ცვლადებისათვის შემოვიტანოთ აღნიშვნა $C_i = (c_{i,4}c_{i,3}c_{i,2}c_{i,1}c_{i,0})_2$ და ყოველ ასეთ $c_{i,j}$ ცვლადს ვუწოდოთ 2^{i+j} რიგის. აქედან გამომდინარე, გვექნება 1 ცალი $2^0 = 1$ რიგის, 2 ცალი 2^1 რიგის, სამი 2^2 რიგის, ოთხი 2^3 რიგის, ხუთი 2^4 რიგის, ისევე ოთხი 2^5 რიგის, სამი 2^6 რიგის, ორი 2^7 რიგის და ერთი 2^8 რიგის ცვლადი.
$c_{0,0}$ ცვლადი პირდაპირ უნდა გადავიდეს, როგორც საბოლოო პასუხის ყველაზე დაბალი ბიტი: $c_0 = c_{0,0}$ (2^0 რიგის ცვლადი მხოლოდ ერთია, ასე რომ, მას არაფერი ემატება).
2^1 რიგის ცვლადები უნდა შევკრიბოთ, შედეგად ვიღებთ ერთ 2^1 რიგის პასუხს (მათ ორობით ჯამს) და ერთ 2^2 რიგის პასუხს (დახსომებულ ბიტს, რომელიც შემდეგში უფრო მაღალი ბიტების ჯამს დაემატება).
ანალოგიურად ვაჯამებთ 2^2 რიგის სამ ცვლადს, პასუხად ვიღებთ ერთ 2^2 რიგის და ერთ 2^3 რიგის ბიტს.
ამ წესით ვაჯამებთ 2^i რიგის ცვლადებს (ორს ან სამს ერთად) და შედეგად ვიღებთ 2^i და 2^{i+1} რიგის ბიტებს, რომლებიც შემდეგ ბიჯში უნდა დავაჯგუფოთ და იგივე წესით ავჯამოთ.
ამ პროცესს ვიმეორებთ მანამ, სანამ არ მივიღებთ ყოველი რიგში ორ ან ერთ ბიტს. ბოლოს ჩვეულებრივი შემკრებით ვაჯამებთ იმ ნაწილს, რომელიც ყოველი რიგის ორ-ორი ბიტისაგან შედგება.
ყოველივე ეს სქემატურად ნაჩვენებია ნახაზში 7.8.
აღსანიშნავია, რომ *HA* ორი ბიტის შემკრები სქემაა, რომელიც a და b ერთ ბიტიანი რიცხვების ორობით ჯამს და დახსომებულ ბიტს გამოითვლის. ფაქტიურად ეს იგივე *FA* სქემაა, სადაც $C = 0$.

სავარჯიშო 7.29: *FA* სქემის გამოყენებით დახაზეთ *HA* სქემა.

ზემოთ მოყვანილ მეთოდს ვოლესის ხე ეწოდება, რადგან მის სქემას ხის სტრუქტურა აქვს: ყოველ შრეში შესაკრებთა რაოდენობა იკლებს. უხეშად რომ დავითვალოთ, სამი შესაკრები ორზე დადის.
მისი ძირითადი იდეაც ესაა: *HA* სქემის გამოყენებით სამი შესაკრები a, b, c ორ ისეთ შესაკრებზე x, y დავიყვანოთ, რომ $a + b + c = x + 2y$.

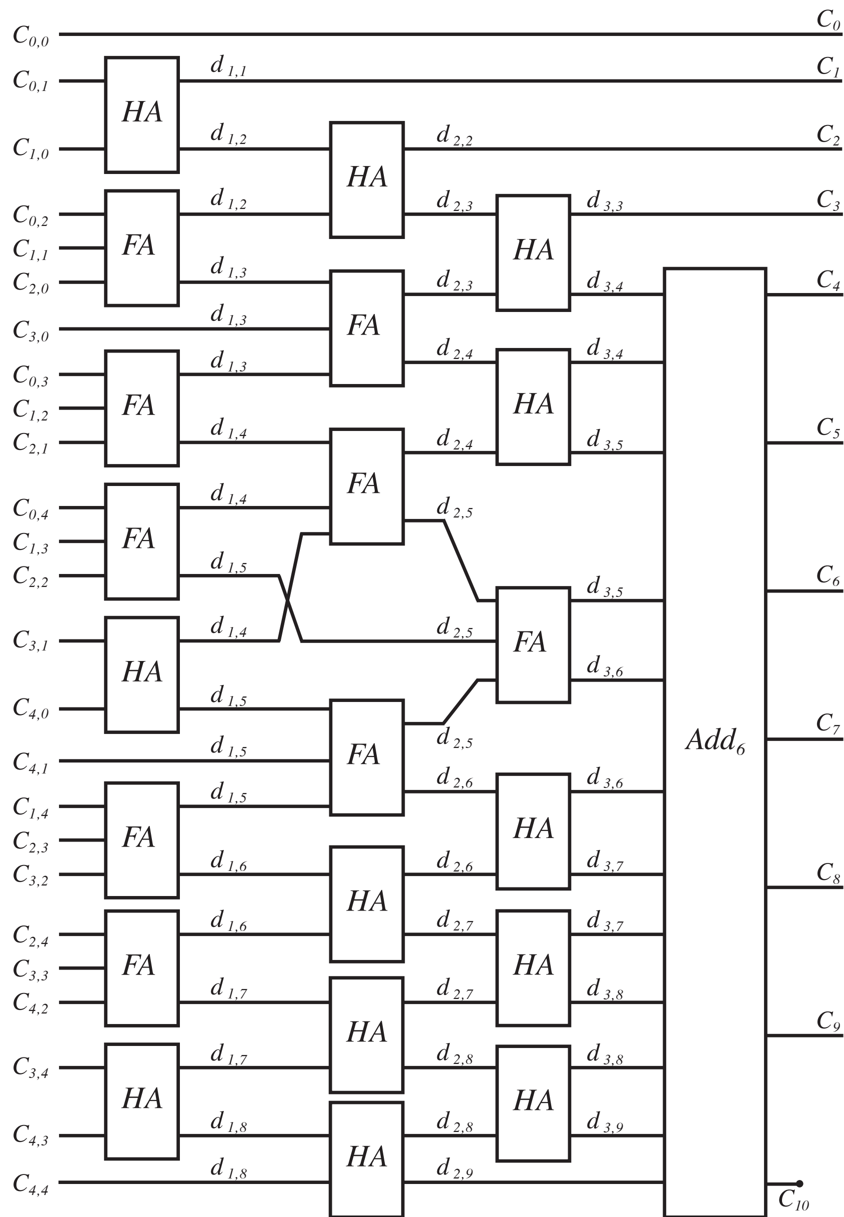
სავარჯიშო 7.30: მაქსიმუმ რამდენ ბიტიანი რიცხვი შეიძლება მივიღოთ ორი n ბიტიანი რიცხვის გამრავლების შედეგად?

7.4.3 კარაცუბა-ოფმანის გამრავლების მეთოდი

1960ან წლებში მოსკოვში მომუშავე მათემატიკოსებმა ანატოლი კარაცუბამ და იური ოფმანმა გამრავლების მეთოდი შეიმუშავეს, რომელმაც ოპერაციათა რაოდენობის ზედა ზღვარი $O(n^2)$-ზე უკეთესია, რითაც მნიშვნელოვანი ნაბიჯი გადადგეს ეფექტური ალგორითმების შემუშავების თვალსაზრისით.

ძირითადი იდეა მარტივია: თუ მოცემულია ორი n ბიტიანი რიცხვი $A = (a_{n-1}...a_0)_2$, $B = (b_{n-1}...b_0)_2$ და ვეძებთ $C = (c_{2n-1}...c_0)_2 = A \cdot B$, ჯერ მონაცემები ორ ტოლ ნაწილად დავყოთ: $A = (A_1A_0)_2$, $B = (B_1B_0)_2$, სადაც $A_1 = (a_{n-1}...a_{\frac{n}{2}})_2$, $A_0 = (a_{\frac{n}{2}-1}...a_0)_2$, $B_1 = (b_{n-1}...b_{\frac{n}{2}})_2$, $B_0 = (b_{\frac{n}{2}-1}...b_0)_2$.
მივიღებთ $A = 2^{\frac{n}{2}}A_1 + A_0$, $B = 2^{\frac{n}{2}}B_1 + B_0$ და

$$
\begin{aligned}
A \cdot B &= (2^{\frac{n}{2}}A_1 + A_0)(2^{\frac{n}{2}}B_1 + B_0) = \\
&= 2^n \cdot A_1B_1 + 2^{\frac{n}{2}}(A_0B_1 + A_1B_0) + A_0B_0.
\end{aligned}
$$



ნახ. 7.8: ორი 5 ბიტიანი რიცხვის გამრავლების ვოლესის ხის შემკრები ნაწილი

როგორც ვხედავთ, ჩასატარებელია 4 გამრავლება, ოღონდ უკვე $\frac{n}{2}$ ბიტიანი რიცხვების. თუ გამრავლების ალგორითმს აღვნიშნავთ როგორც $Mult_n$ და მას რეკურსიულად გამოვიყენებთ, მივიღებთ ელემენტების რაოდენობის შემდეგ შეფასებას:

$$C(Mult_n) = 4 \cdot C(Mult_{\frac{n}{2}}) + 3 \cdot C(Add_{\frac{n}{2}}) \in O(4^{\log n}) = O(n^{\log 4}) = O(n^2)$$

და, როგორც ვხედავთ, ელემენტების რაოდენობას *ვერ* ვამცირებთ. ეს კი იმითაა გამოწვეული, რომ ზემოთ მოყვანილ გამოსახულებაში 4 გამრავლება გვხვდება. თუ რამენაირად მათ შემცირებას მოვახერხებთ, ელემენტების რაოდენობის ზედა ზღვარს კვადრატულზე უკეთესს გავხდით.

სავარჯიშო 7.31: დაამტკიცეთ, რომ $C(Mult_n) = 4 \cdot C(Mult_{\frac{n}{2}}) + 3 \cdot C(Add_{\frac{n}{2}}) \in O(4^{\log n})$.

კარაცუბამ და ოფმანმა შემდეგი ძირითადი იდეა წამოაყენეს: ზედა გამრავლების ფორმულაში $A_1 \cdot B_1$ და $A_0 \cdot B_0$ ნამრავლს გვერდს ვერ ავუვლით. ამიტომ ფრჩხილებში მოცემულ გამოსახულება უნდა შევცვალოთ ისეთით, რომელიც ერთ ახალ გამრავლებასა და $A_1 \cdot B_1$ და $A_0 \cdot B_0$ ელემენტებს შეიცავს.
მისი გამოთვლა შემდეგნაირად შეიძლება:
$A_0B_1 + A_1B_0 = X - A_1 \cdot B_1 - A_0 \cdot B_0$. აქედან გამომდინარე,

$$X = A_0 \cdot B_1 + A_1 \cdot B_0 + A_1 \cdot B_1 + A_0 \cdot B_0 = A_0(B_0 + B_1) + A_1(B_0 + B_1) = (A_1 + A_0) \cdot (B_1 + B_0).$$

საბოლოოდ ვიღებთ ნამრავლის ფორმულას:

$$A \cdot B = 2^n \cdot A_1 \cdot B_1 + 2^{\frac{n}{2}}((A_1 + A_0) \cdot (B_1 + B_0) - A_1 \cdot B_1 - A_0 \cdot B_0) + A_0 \cdot B_0.$$

ერთი შეხედვით გამოსახულება უფრო გართულდა, მაგრამ იმის გამო, რომ $A_1 \cdot B_1$ და $A_0 \cdot B_0$ ერთხელ უკვე გამოვითვალეთ, ფრჩხილებში მყოფ გამოსახულებაში მისი ახლად გამოთვლა საჭირო აღარაა, რადგან აქ მისი ადრე გამოთვლილი მნიშვნელობის გამოყენებაა შესაძლებელი.

საბოლოდ ვიღებთ ელემენტთა რაოდენობის შემდეგ შეფასებას:

$$C(Mult_n) = 3 \cdot C(Mult_{\frac{n}{2}}) + 4 \cdot C(Add_{\frac{n}{2}}) + 2 \cdot C(Sub_{\frac{n}{2}})$$

რადგან $C(Add_k) = C(Sub_k) = const \cdot k$ (როგორც ვნახეთ, ეს ორი ოპერაცია ერთი და იგივე სქემით შეგვიძლია ჩავატაროთ), ვიღებთ:

$$C(Mult_n) = 3 \cdot C(Mult_{\frac{n}{2}}) + 6 \cdot C(Add_{\frac{n}{2}}) = 3 \cdot C(Mult_{\frac{n}{2}}) + const \cdot n \in O(3^{\log n}) = O(n^{\log 3}).$$

ამით დამტკიცდა, რომ შესაძლებელია გამრავლების ოპერაციის კვადრატულზე უკეთეს ელემენტების რაოდენობით ჩატარება, რაც თავის დროზე ძალიან მნიშვნელოვანი შედეგი იყო.

<u>სავარჯიშო 7.32:</u> დაამტკიცეთ, რომ $3 \cdot C(Mult_{\frac{n}{2}}) + const \cdot n \in O(n^{\log 3})$.

თავი 8

დინამიკური პროგრამირება

8.1 ფიბონაჩის რიცხვების მოძებნა

განვიხილოთ ამოცანა: დაწერეთ პროგრამა, რომელიც გაარკვევს რამდენნაირი განსხვავებული გზით შეიძლება კიბის მე-20 საფეხურზე ასვლა, თუკი ყოველ სვლაზე შესაძლებელია ერთი ან ორი საფეხურით გადაადგილება. ცხადია, რომ ნებისმიერ i-ურ საფეხურზე შესაძლებელია მოვხვდეთ მხოლოდ წინა ორი i-1 და i-2 საფეხურიდან. თუკი გვეცოდინება, რამდენი განსხვავებული გზით შეიძლება მოვხვდეთ ამ ორ საფეხურზე, მაშინ i-ურ საფეხურზე მისვლის ვარიანტების რაოდენობა მათი ჯამი იქნება. ასეთი დასკვნის საფუძველს გვაძლევს ის ფაქტი, რომ i-2 საფეხურიდან - ორ საფეხურზე, ხოლო i-1 საფეხურიდან ერთ საფეხურზე ანაცვლებით ავალთ i-ურ საფეხურზე. ის ვარიანტი, რომლითაც i-2 საფეხურიდან i-1-ზე შეიძლება ასვლა, უკვე გათვალისწინებული იქნება i-1-ე საფეხურზე ასვლის ვარიანტთა რაოდენობაში. მაშასადამე, ადგილი აქვს რეკურენტულ ფორმულას:

$$RAOD(i) = RAOD(i-1) + RAOD(i-2)$$

რადგან პირველ საფეხურზე მხოლოდ ერთი გზით შეიძლება მოხვედრა, ხოლო მეორეზე - ორი გზით (პირდაპირ ორ საფეხურზე ანაცვლებით და პირველი საფეხურიდან ერთ საფეხურზე ანაცვლებით), შეგვიძლია განვსაზღვროთ, რომ RAOD(1)=1 და RAOD(2)=2. ყველა დანარჩენი წევრის გამოსათვლელად კი თანმიმდევრულად შევავსოთ ერთგანზომილებიანი 17-ელემენტიანი მასივი:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584

შევნიშნოთ, რომ მიიღება ფიბონაჩის მიმდევრობა, რომელსაც პირველი წევრი აკლია. ჩვენს მიერ რეკურსიული გზით ამოხსნილი ამოცანა, შედარებით მარტივი გზით ამოიხსნა და ამასთანავე, ამოცანის მუშაობის დრო ამოცანის ზომის პროპორციულია, რადგან საბოლოო პასუხამდე საშუალედო მნიშვნელობები მასივში დავიმახსოვრეთ. ამოცანების ამოხსნის ასეთ მეთოდს, როდესაც ყოველ ბიჯზე თვლის დროს მიღებული საშუალედო შედეგების დამახსოვრება ხდება სპეციალურ ცხრილში და ეს შედეგები გამოიყენება მომდევნო ბიჯების გამოთვლისას, უწოდებენ დინამიკური პროგრამირების მეთოდს.
ჩვენს მიერ გამოყენებულ მეთოდს ჰქონდა დამატებითი სპეციფიკა, რადგან ამ ფუნქციის მოცემული მნიშვნელობის გამოსათვლელად ჩვენ ასევე გამოვთვალეთ ყველა საშუალედო მნიშვნელობა, დაწყებული საბაზოდან (i=1 და i=2), და ყოველ ჯერზე ადრე გამოთვლილ მნიშვნელობებს ვიყენებდით მიმდინარე მნიშვნელობის გამოსათვლელად. ამ ტექნოლოგიას ეწოდება **აღმავალი დინამიკური პროგრამირება** (bottom-up dynamic programming).
დაღმავალი დინამიკური პროგრამირება (top-down dynamic programming) კიდევ უფრო მარტივი ტექნოლოგიაა, რაც საშუალებას იძლევა ავტომატურად შესრულდეს რეკურსიული ფუნქციები იტერაციათა იგივე (ან უფრო მცირე) რაოდენობისას, რაც საჭიროა აღმავალი დინამიკური დაპროგრამების შემთხვევაში. ამასთან, რეკურსიულ პროგრამაში უნდა ჩაიდოს ინსტრუმენტალური საშუალებები, რომ დავიმახსოვროთ საშუალედო შედეგები და შემდეგ შევამოწმოთ შენახული მნიშვნელობების არსებობა რომელიმე მონაცემის ხელმეორედ გამოთვლის თავიდან აცილების მიზნით.

8.2 დინამიკური პროგრამირების ამოცანების სპეციფიკა

დინამიკური პროგრამირების ზოგადი პრინციპები პირველად აღწერა ამერიკელმა მათემატიკოსმა რიჩარდ ბელმანმა მეოცე საუკუნის 50-იან წლებში და ეს მეთოდი დღეისათვის წარმოადგენს ერთ-ერთ ყველაზე მძლავრ საშუალებას სხვადასხვა სახის ამოცანების ამოსახსნელად.

ტიპურ შემთხვევებში, დინამიკური პროგრამირების მეთოდი გამოიყენება ოპტიმიზაციის ამოცანების ამოსახსნელად. რა თვისებები უნდა ჰქონდეს ამოცანას, რა ტიპის ამოცანებისთვისაა შესაძლებელი ამ მეთოდის გამოყენება? **ოპტიმალური ქვესტრუქტურა** (optimal substructure) და **ქვეამოცანების გადაფარვა** (overlapping subproblems) არის ორი ძირითადი ნიშანი, რითიც უნდა ხასიათდებოდეს ამოცანა, რომ მისთვის გამოყენებულ იქნას დინამიკური პროგრამირების მეთოდი.
ამბობენ, რომ ამოცანას აქვს **ოპტიმალური ქვესტრუქტურა**, თუ ამოცანის ოპტიმალური ამონახსნი შეიცავს მისი რომელიმე (ან რამდენიმე) ქვეამოცანის ოპტიმალურ ამონახსნს. იმისათვის, რომ დავრწმუნდეთ, რომ ამოცანას აქვს ეს თვისება, უნდა ვაჩვენოთ, რომ ქვეამოცანის ამონახსნის გაუმჯობესება აუმჯობესებს საწყისი ამოცანის ამონახსნსაც. **ქვეამოცანების გადაფარვის** თვისება ამოცანისთვის ნიშნავს, რომ მას არა აქვს ქვეამოცანების "დიდი რაოდენობა", იმ აზრით, რომ რეკურსიული ალგორითმით ხდება ერთი და იგივე ქვეამოცანების ამოხსნა და არ წარმოიქმნება ახალი ქვეამოცანები. ეს საშუალებას იძლევა ქვეამოცანა ამოიხსნას მხოლოდ ერთხელ და მოხდეს მისი დამახსოვრება.
დინამიკური პროგრამირების მეთოდით ამოცანის ამოხსნის პროცესი შეიძლება დავყოთ 4 ეტაპად:

1. ოპტიმალური ამონახსნის სტრუქტურის აღწერა
2. რეკურენტული თანაფარდობის პოვნა ქვეამოცანებსა და ოპტიმალურ ამონახსნს შორის
3. აღმავალი დინამიკური პროგრამირების გამოყენებით, ქვეამოცანების ოპტიმალური მნიშვნელობების გამოთვლა
4. წინა ეტაპებზე მიღებული ინფორმაციის საფუძველზე ოპტიმალური ამონახსნის აგება

ოპტიმალური ამონახსნის სტრუქტურის აღწერის შედეგად დავადგენთ კავშირს ქვეამოცანებისა და საწყისი ამოცანის ოპტიმალურ ამონახსნებს შორის, შემდეგ, ქვეამოცანების ოპტიმალური ამონახსნების საშუალებით, ავაგებთ საწყისი ამოცანის ამონახსნს. **რეკურენტული თანაფარდობის აღწერით** დადგინდება ოპტიმალური ამოცანის ღირებულება ქვეამოცანების ოპტიმალური ამონახსნების ტერმინებში. თუ ამოცანა აკმაყოფილებს ქვეამოცანების გადაფარვის თვისებას და მრავალგზის გვიხდება ერთი და იგივე ქვეამოცანის ამოხსნა, ასეთ დროს ძირითადი ტექნიკური საშუალებაა - დავიმახსოვროთ ქვეამოცანის ამონახსნები იმ შემთხვევისათვის, როცა ისინი ისევ შეგვხვდება. მათ ამოსახსნელად გამოვიყენოთ აღმავალი დინამიკური პროგრამირების მეთოდი. ბოლოს, არსებული ინფორმაციის საფუძველზე ავაგებთ ოპტიმალურ ამონახსნს.

8.3 უგრძესი გზის პოვნა რიცხვების სამკუთხა ცხრილში

განვიხილოთ სურ. 8.1-ზე გამოსახულია რიცხვებისაგან აგებული სამკუთხედი. დაწერეთ პროგრამა, რომელიც იპოვის სამკუთხედის ზედა წვეროდან დაწყებულ და მის ფუძეზე დამთავრებულ გზების სიგრძეებს შორის მაქსიმალურს. გზის სიგრძედ ითვლება მასზე განლაგებული რიცხვების ჯამი.

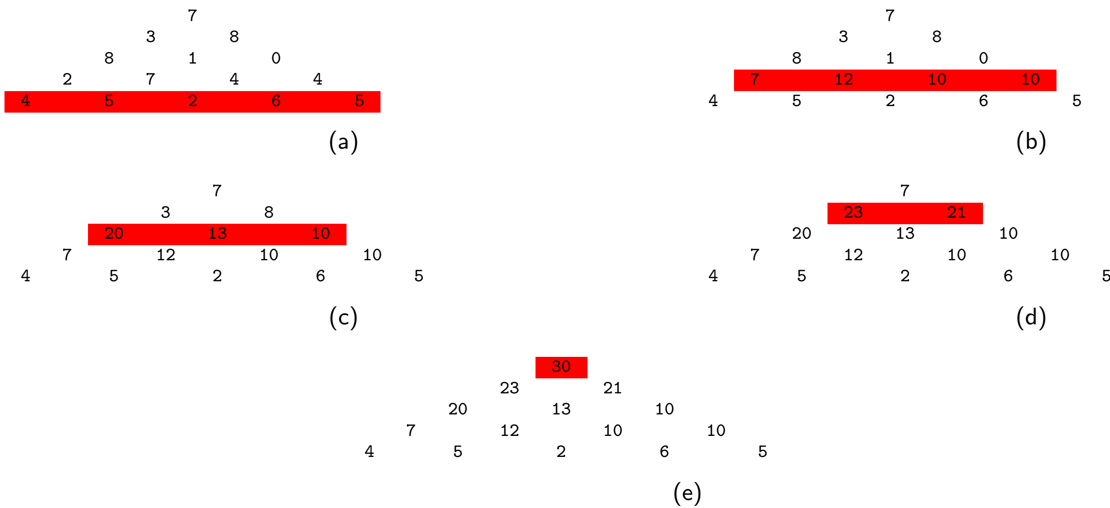
```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

ნახ. 8.1

- ყოველი ბიჯი გზაზე კეთდება დიაგონალურად ქვემოთ და მარცხნივ ან დიაგონალურად ქვემოთ და მარჯვნივ
- სტრიქონთა რაოდენობა სამკუთხედში მეტია 1-ზე და ნაკლებია ან ტოლია 1000-ზე
- სამკუთხედი შედგება მთელი რიცხვებისაგან 0-დან 99-მდე

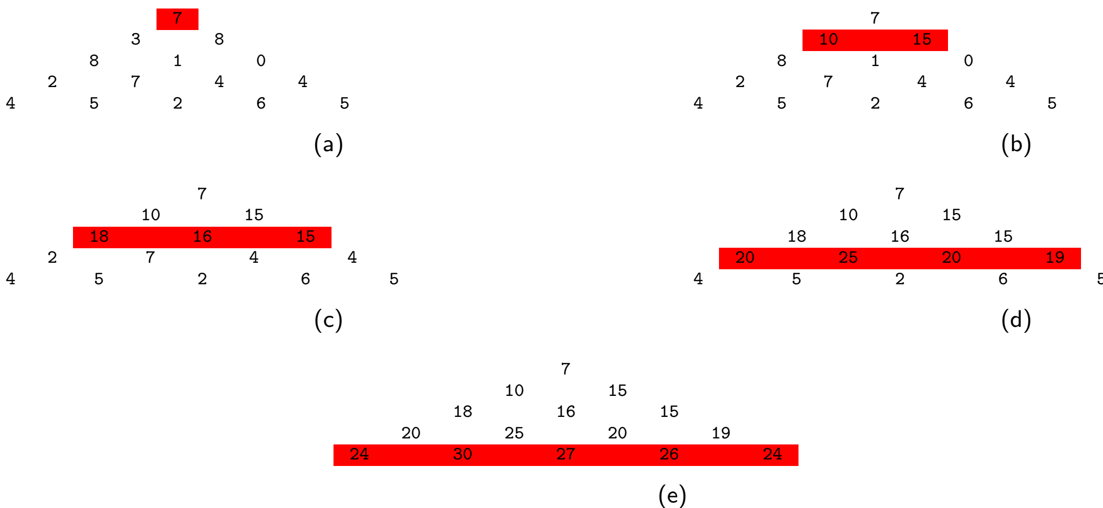
დიდი მოცულობის შემავალი მონაცემებისათვის მათი რეალიზაცია შესაძლებელია ერთგანზომილებიან მასივშიც, მაგრამ ამოცანაში მოცემული პარამეტრებისათვის ორგანზომილებიანი მასივის (ზომით 1000×1000) გამოყენებაც შეიძლება. მართალია, ასეთ მასივში ელემენტების დიდი ნაწილი NIL მნიშვნელობის იქნება (ჩვენი ამოცანისთვის გამოდგებოდა, ვთქვათ -1) და მანქანურ მეხსიერებას უსარგებლოდ დაიკავებს, სამაგიეროდ თავიდან ავიცილებთ მეზობელი ელემენტების ფორმულებით გამოთვლას, რაც ერთგანზომილებიან მასივშია საჭირო.

მოცემული მაგალითისათვის, ვთქვათ, ჩვენ უკვე შევარჩიეთ მაქსიმალური გზა პირველ ოთხ სტრიქონში და მხოლოდ მეხუთე სტრიქონში დაგვრჩა რიცხვი ასარჩევი. თუკი ჩვენ ვდგავართ მეოთხე სტრიქონის პირველ ელემენტზე (რიცხვი 2), მაშინ უნდა ავირჩიოთ 4-სა და 5-ს შორის უდიდესი და მისკენ გავაგრძელოთ მოძრაობა, თუკი ვდგავართ მეოთხე სტრიქონის მეორე ელემენტზე (რიცხვი 7), უნდა ავირჩიოთ უდიდესი 5-სა და 2-ს შორის და მასზე გადავიდეთ და ა.შ. ცხადია, რომ ბოლოსწინა სტრიქონიდან სვლის გაკეთებისას ნებისმიერ შემთხვევაში ორ ქვედა მეზობელს შორის უდიდესი უნდა ავირჩიოთ. აქედან გამომდინარეობს, რომ თუკი ბოლოსწინა სტრიქონის თითოეულ ელემენტს წინასწარ დავუმატებთ ორ ქვედა მეზობელთაგან უდიდესს და ბოლო სტრიქონს საერთოდ გავაუქმებთ - მივიღებთ იმავე მაქსიმალური სიგრძის მქონე სამკუთხედს, როგორიც თავდაპირველად გვქონდა მოცემული. თუკი ახალი სამკუთხედისთვისაც გავიმეორებთ იგივე ქმედებას, სამკუთხედის ზომა კიდევ ერთი სტრიქონით შემცირდება და ასე გავაგრძელებთ მანამ, სანამ არ დაგვრჩება მხოლოდ ზედა წვერო, რომელშიც უკანასკნელ ბიჯზე ჩაწერილი რიცხვი წარმოადგენს ჩვენი ამოცანის ამონახსნს.



ნახ. 8.2

ანალოგიურ შედეგს მივიღებდით, თუკი ვიმოძრავებდით სამკუთხედის ზედა წვეროდან ფუძისაკენ შემდეგი პრინციპით: თუ მეორე სტრიქონში მოთავსებულ ელემენტს ჰყავს მხოლოდ ერთი ზედა მეზობელი, ამ უკანასკნელის მნიშვნელობას ვუმატებთ მას, ხოლო თუ ორი ზედა მეზობელი ჰყავს - ვუმატებთ მათ შორის უდიდესს. როცა მეორე სტრიქონის ყველა ელემენტს ასე გარდავქმნით, ვაუქმებთ პირველ სტრიქონს და პროცესს ვიმეორებთ. სურ. 8.3-ზე მოცემულია ამ ალგორითმის მუშაობა (საწყისი სამკუთხედი იგივეა):



ნახ. 8.3

განსხვავებით პირველი ალგორითმისაგან (როცა ფუძიდან წვეროსაკენ ვმოძრაობდით) აქ პირდაპირ პასუხს ვერ ვღებულობთ და საჭიროა მაქსიმალური ელემენტის პოვნა უკანასკნელ სტრიქონში. ორივე ალგორითმში იკარგება

მოძრაობის მარშრუტი და მის აღსადგენად საჭიროა შედეგამდე განვლილ გზაზე უკან დაბრუნება და ყოველ ბიჯზე იმის შემოწმება, თუ რომელი წვერიდან მოვედით მოცემულ ელემენტზე.
ანალოგიური ალგორითმით - ყოველ ბიჯზე მინიმალური ელემენტების შეკრებით, შესაძლებელია მინიმალური ჯამის პოვნაც.

8.4 უდიდესი საერთო ქვემიმდევრობის პოვნა

დინამიკური პროგრამირების კლასიკურ ნიმუშს წარმოადგენს ე.წ. უდიდესი საერთო ქვემიმდევრობის პოვნის ამოცანა:
ვიტყვით, რომ მოცემული მიმდევრობიდან მივიღეთ ქვემიმდევრობა, თუ მიმდევრობის ზოგიერთი ელემენტი წავშალეთ, ხოლო დარჩენილ ელემენტებზე შენარჩუნებულია რიგი (თავად მიმდევრობაც ითვლება საკუთარ ქვემიმდევრობად). მათემატიკური ფორმულირებით: $Z = (z_1, z_2, \ldots, z_k)$ მიმდევრობას ეწოდება $X = (x_1, x_2, \ldots, x_n)$ მიმდევრობის ქვემიმდევრობა (subsequence), თუ არსებობს $(i_1, i_2, \ldots, i_k)$ ინდექსთა მკაცრად ზრდადი მიმდევრობა, რომლისთვისაც $z_j = x_{i_j}$, ყველა $j = 1, 2, \ldots, k$-სათვის. მაგალითად, $Z = (B, C, D, B)$ არის $X = (A, B, C, B, D, A, B)$ მიმდევრობის ქვემიმდევრობა. ინდექსთა შესაბამისი მიმდევრობაა $(2, 3, 5, 7)$. შევნიშნოთ, რომ მათემატიკისაგან განსხვავებით, ლაპარაკია მხოლოდ სასრულ მიმდევრობებზე.
ვიტყვით, რომ Z მიმდევრობა წარმოადგენს X და Y მიმდევრობების საერთო ქვემიმდევრობას (common subsequence), თუ Z წარმოადგენს როგორც X-ის, ასევე Y-ის ქვემიმდევრობას. მაგალითად, $X = (A, B, C, B, D, A, B)$, $Y = (B, D, C, A, B, A)$, $Z = (B, C, A)$. ამ მაგალითში Z მიმდევრობა არ არის უდიდესი X-ისა და Y-ის საერთო ქვემიმდევრობათა შორის - $Z = (B, C, B, A)$ მიმდევრობა უფრო გრძელია. თავად (B, C, B, A) მიმდევრობა კი უდიდესს წარმოადგენს X-ისა და Y-ის საერთო ქვემიმდევრობათა შორის, რადგან 5 სიგრძის მქონე საერთო ქვემიმდევრობა არ არსებობს. უდიდესი საერთო ქვემიმდევრობა შეიძლება რამდენიმე იყოს, მაგ.: (B, D, A, B) სხვა უდიდესი საერთო ქვემიმდევრობაა მოცემული X-ისა და Y-სათვის.
უდიდესი საერთო ქვემიმდევრობის (შემოკლებით **უსქ**, LCS=longest-common-subsequence) ამოცანა მდგომარეობს იმაში, რომ მოცემული X და Y მიმდევრობებისათვის ვიპოვოთ უდიდესი სიგრძის მქონე საერთო ქვემიმდევრობა. ეს ამოცანა იხსნება დინამიკური პროგრამირების მეთოდით.
თუკი **უსქ**-ის ამოცანას ამოვხსნით სრული გადარჩევით, ალგორითმს დასჭირდება ექსპონენციალური დრო, რადგან m სიგრძის მიმდევრობა შეიცავს 2^m ქვემიმდევრობას (იმდენივეს, რამდენ ქვესიმრავლესაც შეიცავს $\{1, 2, \ldots, m\}$ სიმრავლე). მაგრამ როგორც ქვემოთ მოყვანილი თეორემა გვიჩვენებს, უსქ-ის ამოცანას აქვს ოპტიმალურობის თვისება ქვეამოცანებისათვის. ქვეამოცანებად შეგვიძლია განვიხილოთ მოცემული ორი მიმდევრობის პრეფიქსების წყვილთა სიმრავლეები. ვთქვათ, $X = (x_1, x_2, \ldots, x_m)$ რაღაც მიმდევრობაა. მისი i სიგრძის პრეფიქსი არის მიმდევრობა $X = (x_1, x_2, \ldots, x_i)$, სადაც i მოთავსებულია 0-დან m-მდე. მაგალითად, თუ $X = (A, B, C, B, D, A, B)$, მაშინ $X_4 = (A, B, C, B)$, ხოლო X_0 ცარიელი ქვემიმდევრობაა.

თეორემა 8.1. (უსქ-ის აგებულების შესახებ). ვთქვათ $Z = (z_1, z_2, \ldots, z_k)$ ერთ-ერთი უდიდესი საერთო ქვემიმდევრობაა $X = (x_1, x_2, \ldots, x_m)$ და $Y = (y_1, y_2, \ldots, y_n)$ მიმდევრობებისათვის. მაშინ:

1. თუ $x_m = y_n$, მაშინ $z_k = x_m = y_n$ და Z_{k-1} წარმოადგენს უსქ-ის X_{m-1}-ისა და Y_{n-1}-სათვის
2. თუ $x_m \neq y_n$ და $z_k = x_m$ მაშინ Z წარმოადგენს უსქ-ის X_{m-1}-ისა და Y-სათვის
3. თუ $x_m = y_n$ და $z_k = y_n$ მაშინ Z წარმოადგენს უსქ-ის X-ისა და Y_{n-1}-სათვის.

Proof.

1. ვთქვათ, $x_m = y_n$. რომ სრულდებოდეს $z_k = x_m$, მაშინ Z მიმდევრობისთვის $x_m = y_n$ ელემენტის მიმატებით, მივიღებდით X და Y მიმდევრობების $k+1$ სიგრძის საერთო ქვემიმდევრობას, რაც ეწინააღმდეგება პირობას. ე.ი. $z_k = x_m = y_n$. ამიტომ, Z_{k-1} არის $k-1$ სიგრძის X_{m-1}-ს და Y_{n-1}-ს საერთო ქვემიმდევრობა. ვაჩვენოთ, რომ ის უდიდესი საერთო ქვემიმდევრობაა. დავუშვათ საწინააღმდეგო, ვთქვათ, არსებობს X_{m-1} და Y_{n-1} მიმდევრობების, $(k-1)$-ზე გრძელი საერთო ქვემიმდევრობა, მაშინ თუ მას მივუწერთ $x_m = y_n$ ელემენტს, მივიღებთ X და Y მიმდევრობების საერთო ქვემიმდევრობას, რომელიც k-ზე გრძელია, რასაც მივყავართ წინააღმდეგობამდე.
2. ვთქვათ, $x_m \neq y_n$, რადგან $z_k \neq x_m$ ამიტომ Z წარმოადგენს X_{m-1}-ს და Y-ს საერთო ქვემიმდევრობას. ვაჩვენოთ, რომ ის უდიდესი საერთო ქვემიმდევრობაა. რომ არსებობდეს X_{m-1}-ს და Y-ს k-ზე გრძელი საერთო ქვემიმდევრობა, მაშინ ის, აგრეთვე, იქნებოდა X-ს და Y-ს საერთო ქვემიმდევრობა, რაც ეწინააღმდეგება იმას, რომ Z არის X-ს და Y-ს უსქ.
3. მტკიცება 2-ს ანალოგიურად.

□

ამ თეორემიდან ჩანს, რომ ორი მიმდევრობის უსქ შეიცავს მათივე პრეფიქსების უსქ-ს. აქედან შეიძლება დავასკვნათ, რომ უსქ-ის ამოცანას აქვს ოპტიმალური ქვესტრუქტურა. როგორც ქვემოთ ვნახავთ, ადგილი აქვს ქვეამოცანების გადაფარვასაც.

თეორემა 6.1 გვიჩვენებს, რომ უსქ-ის პოვნა $X = (x_1, x_2, \ldots, x_m)$ და $Y = (y_1, y_2, \ldots, y_n)$ მიმდევრობებისათვის დადის ან ერთი, ან ორი ქვეამოცანის ამოხსნაზე. თუ $x_m = y_n$, მაშინ საკმარისია ვიპოვოთ X_{m-1}-ისა და Y_{n-1}-ის უსქ და მას მივუწეროთ $x_m = y_n$. თუ $x_m \neq y_n$, მაშინ უნდა ამოიხსნას ორი ქვეამოცანა: ვიპოვოთ უსქ X_{m-1}-ისა და Y-სათვის, ვიპოვოთ ასევე უსქ X-ისა და Y_{n-1}-სათვის და მათ შორის უდიდესი იქნება X-ისა და Y-ის უსქ.

ზემოთ თქმულიდან ცხადი ხდება, რომ წარმოიქმნება ქვეამოცანების გადაფარვა, რადგან რომ ვიპოვოთ X-ისა და Y-ის უსქ, ჩვენ შეიძლება დაგვჭირდეს X_{m-1}-ისა და Y-ის, ასევე X-ისა და Y_{n-1}-ის უსქ-ების პოვნა, თითოეული ამ ამოცანიდან შეიცავს X_{m-1}-ისა და Y_{n-1}-ის უსქ-ის პოვნის ქვეამოცანას. ანალოგიური გადაფარვები შეგვხვდება სხვა შემთხვევებშიც.

ავაგოთ რეკურენტული თანაფარდობა ოპტიმალური ამონახსნის ღირებულებისათვის. $c[i, j]$-თი აღვნიშნოთ უსქ-ის სიგრძე X_i და Y_j მიმდევრობებისათვის. თუ i ან j ტოლია 0-ის, მაშინ ორი მიმდევრობიდან ერთ-ერთი ცარიელია და $c[i, j] = 0$. ზემოთ თქმული შეიძლება ასე ჩაიწეროს:

$$c[i,j] = \begin{cases} 0 & \text{თუ } i = 0 \text{ ან } j = 0 \\ c[i-1, j-1] + 1 & \text{თუ } i, j > 0 \text{ და } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{თუ } i, j > 0 \text{ და } x_i \neq y_j \end{cases}$$

რეკურენტული თანაფარდობიდან გამომდინარე, სირთულეს არ წარმოადგენს რეკურსიული ალგორითმის დაწერა, რომელიც ექსპონენციალურ დროში მოძებნიდა უსქ-ის სიგრძეს ორი მოცემული მიმდევრობისათვის. მაგრამ რადგან განსხვავებული ქვეამოცანების რაოდენობა mn-ს პროპორციულია, უმჯობესია დინამიკური პროგრამირების გამოყენება.

LCS-LENGTH ალგორითმის შემავალი მონაცემებია $X = (x_1, x_2, \ldots, x_m)$ და $Y = (y_1, y_2, \ldots, y_n)$ მიმდევრობები. $c[i, j]$ რიცხვები ჩაიწერება $c[0..m, 0..n]$ ცხრილში შემდეგნაირად: ჯერ შეივსება მარცხნიდან მარჯვნივ პირველი სტრიქონი, შემდეგ მეორე და ა.შ. გარდა ამისა ალგორითმი $b[1..m, 1..n]$ ცხრილში იმახსოვრებს $c[i, j]$-ის "წარმომავლობას". $b[i, j]$ უჯრედში შეიტანება ისარი, რომელიც მიუთითებს შემდეგი სამი უჯრედიდან ერთ-ერთის კოორდინატებს: $(i-1, j-1)$, $(i-1, j)$ ან $(i, j-1)$, იმისდა მიხედვით თუ რისი ტოლია $c[i, j]$ შესაბამისად - $c[i-1, j-1]+1$, $c[i-1, j]$ თუ $c[i, j-1]$. ალგორითმის მუშაობის შედეგებია c და b ცხრილები. მოვიყვანოთ შესაბამისი ფსევდოკოდი:

Algorithm 19: Longest Common Subsequence Length

Input: სიმბოლოების ორი მიმდევრობა
Output: უდიდესი საერთო ქვემიმდევრობა და მისი სიგრძე

```
1  LCS-LENGTH(X, Y) :
2      m = len(X);
3      n = len(Y);
4      for i=0; i<=m; i++ :
5          c[i][0] = 0;
6      for j=1; j<=n; j++ :
7          c[0][j] = 0;
8      for i=1; i<=m; i++ :
9          for j=1; j<=n; j++ :
10             if X[i] == Y[i] :
11                 c[i][j] = c[i-1][j-1] + 1;
12                 b[i][j] = ↖;
13             elif c[i-1][j] >= c[i][j-1] :
14                 c[i][j] = c[i-1][j];
15                 b[i][j] = ↑;
16             else:
17                 c[i][j] = c[i][j-1];
18                 b[i][j] = ←;
19     return c, b;
```

ქვემოთ ნაჩვენებია LCS-LENGTH ალგორითმის მუშაობა $X =< A, B, C, B, D, A, B >$ და $Y =< B, D, C, A, B, A >$ მიმდევრობებისათვის. b და c ცხრილები გაერთიანებულია და (i, j) კოორდინატების მქონე უჯრედში ჩაწერილია რიცხვი $c[i, j]$ და ისარი $b[i, j]$. რიცხვი 4 ცხრილის ქვედა მარჯვენა უჯრედში წარმოადგენს უსქ-ის სიგრძეს. ამ პასუხის მიღების გზა ნახ. 3-ზე რუხი ფერითაა ნაჩვენები.

	j	0	1	2	3	4	5	6
i		y_i	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0↑	0↑	0↑	1↖	1←	1↖
2	B	0	1↖	1←	1←	1↑	2↖	2←
3	C	0	1↑	1↑	2↖	2←	2↑	2↑
4	B	0	1↖	1↑	2↑	2↑	3↖	3←
5	D	0	1↑	2↖	2↑	2↑	3↑	3↑
6	A	0	1↑	2↑	2↑	3↖	3↑	4↖
7	B	0	1↖	2↑	2↑	3↑	4↖	4↑

ნახ. 8.4

LCS-LENGTH ალგორითმის მუშაობის დროა $O(mn)$. თითოეული უჯრედის შესავსებად საჭიროა $O(1)$ ბიჯი. უსქ-ის სიგრძის პოვნის შემდეგ b ცხრილის საშუალებით შეგვიძლია დავადგინოთ თავად უსქ. ამისათვის საჭიროა $b[m,n]$ უჯრედიდან დავბრუნდეთ უკან და ვიპოვოთ ↖ ისრები. ამის რეალიზაციას ახდენს რეკურსიული პროცედურა PRINT-LCS.

Algorithm 20: Print Longest Common Subsequence

Input: LCS-LENGTH(X,Y)-ით გამოთვლილი b მატრიცა, X სიმბოლოების მიმდევრობა, რეკურსიის გამოძახებისას $i = |X|$ და $j = |Y|$

Output: ბეჭდავს უდიდეს საერთო ქვემიმდევრობას

```
1  PRINT-LCS(b, X, i, j) :
2      if i == 0 or j == 0 :
3          return ;
4      if b[i][j] == ↖ :
5          PRINT-LCS(b, X, i-1, j-1);
6          print(X[i]);
7      elif b[i][j] == ↑ :
8          PRINT-LCS(b, X, i-1, j);
9      else:
10         PRINT-LCS(b, X, i, j-1);
```

მოყვანილი მაგალითისათვის პროცედურა დაბეჭდავს BCBA-ს. პროცედურის მუშაობის დროა O(m+n), რადგან ყოველ ბიჯზე მცირდება ან m, ან n. უსქ-ის ის ამოცანაზე მსჯელობის დასასრულს, შევნიშნოთ, რომ შესაძლებელია ალგორითმის გაუმჯობესებაც. მაგალითად, ჩვენს შემთხვევაში შეიძლებოდა b მასივი საერთოდ არ გამოგვეყენებინა და უსქ c მასივის გადამოწმებით დაგვედგინა. ნებისმიერი c[i][j] რიცხვისათვის მოგვიწევდა შეგვემოწმებინა c[i-1][j], c[i][j-1] და c[i-1][j-1] უჯრედები, რისთვისაც O(1) დროა საჭირო. მაშასადამე, უსქ-ის პოვნა იგივე O(m+n) დროში ერთი ცხრილითაც შეიძლებოდა.

8.5 მატრიცათა მიმდევრობის გადამრავლების ამოცანა

ორი A და B მატრიცა შეიძლება გადამრავლდეს მხოლოდ მაშინ, თუკი A მატრიცის სვეტების რაოდენობა ემთხვევა B მატრიცის სტრიქონების რაოდენობას. თუ A მატრიცის ზომებია $p\times q$ და B მატრიცის ზომებია $q\times r$, მაშინ მათი გადამრავლებით მიიღება $p\times r$ ზომის C მატრიცა. ოპერაციების რაოდენობა გადამრავლების სტანდარტულ ალგორითმში არის pqr-ის პროპორციული, რადგან ნამრავლის ფორმულა შეიცავს სამ ერთმანეთში ჩადგმულ ციკლს. ვთქვათ, საჭიროა n ცალი მატრიცის $A_1, A_2, \ldots, A_n$ ერთმანეთზე გადამრავლება. ამ ამოცანის გადასაწყვეტად წინასწარ საჭიროა ფრჩხილების სრულად განთავსება, რათა განისაზღვროს გამრავლებათა თანმიმდევრობა. ჩვენ ვიტყვით, რომ მატრიცათა ნამრავლში ფრჩხილები სრულადაა განთავსებული, თუ ეს ნამრავლი შედგება ან ერთადერთი მატრიცისაგან, ან არის ფრჩხილებში მოთავსებული, ორი სრულად განთავსებული ფრჩხილების მქონე მატრიცათა ნამრავლის ნამრავლი. მაგალითად ოთხი $A_1A_2A_3A_4$ მატრიცის ნამრავლში ფრჩხილები შესაძლოა ხუთნაირად განთავსდეს:

$$(A_1(A_2(A_3A_4)))\ (A_1((A_2A_3)A_4))\ ((A_1A_2)(A_3A_4))\ ((A_1(A_2A_3))A_4)\ (((A_1A_2)A_3)A_4)$$

რადგან მატრიცების ნამრავლი ასოციაციურია, საბოლოო შედეგი ყოველთვის ერთი და იგივეა, მაგრამ ჩატარებული ოპერაციების რაოდენობის მიხედვით ვარიანტები შეიძლება მკვეთრად განსხვავდებოდნენ. მაგალითად, სამი $< A_1, A_2, A_3 >$ მატრიცა შეიძლება ორნაირად გადავამრავლოთ: $((A_1A_2)A_3)$ და $(A_1(A_2A_3))$. ვთქვათ, მატრიცების ზომებია შესაბამისად 10×100, 100×5 და 5×50. $((A_1A_2)A_3)$ განლაგებით საჭიროა $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ ოპერაცია, ხოლო $(A_1(A_2A_3))$ განლაგებით - $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ ოპერაცია. მაშასადამე, პირველი გზით გამრავლება 10-ჯერ უფრო მომგებიანია.

მატრიცათა მიმდევრობის გადამრავლების ამოცანა (მატრიხ-ცჰაინ მულტიპლიცატიონ პრობლემ) შეიძლება ასე ჩამოყალიბდეს: ვთქვათ, მოცემულია n ცალი მატრიცისაგან შემდგარი $< A_1, A_2, \ldots, A_n >$ მიმდევრობა, რომელთა ზომებიც განსაზღვრულია. (A_i მატრიცის ზომებია $p_{i-1} \times p_i$). საჭიროა მოიძებნოს ფრჩხილების ისეთი სრული განთავსება, რომ მატრიცათა მიმდევრობის გადამრავლებისას შესრულდეს მინიმალური რაოდენობის გამრავლების ოპერაცია. ამ ამოცანის გადასაწყვეტად სრული გადარჩევა არ გამოდგება, რადგან ვარიანტების რაოდენობა ექსპონენციალურადაა დამოკიდებული მატრიცების რაოდენობაზე. ამის დასამტკიცებლად მატრიცების მოცემული მიმდევრობა შეგვიძლია დავყოთ 3-3 წევრიან ჯგუფებად. თითოეულ ჯგუფში ნამრავლის გამოსათვლელად არსებობს ორი ვარიანტი. მაშასადამე $3n$ მატრიცისათვის იარსებებს არანაკლებ 2^n ვარიანტისა. ვცადოთ ამოცანის ამოხსნა დინამიკური პროგრამირების მეთოდით.

გამოყენებადია თუ არა დინამიკური პროგრამირება? აღვწეროთ ოპტიმალურ ამონახსნთა სტრუქტურა. აღვნიშნოთ $A_{i..j}$-ით მატრიცების ნამრავლი $A_iA_{i+1}\ldots A_j$. $A_1A_2\ldots A_n$ ნამრავლში ფრჩხილების ოპტიმალური განთავსება განაპირობებს ისეთი k-ს არსებობას, $(1 \le k < n)$ რომ ყველა მატრიცის ნამრავლის გამოსათვლელად ჩვენ ჯერ ვითვლით $A_{1..k}$ და $A_{k+1..n}$ ნამრავლებს, ხოლო შემდეგ მათ ვამრავლებთ ერთმანეთზე და ვიღებთ ოპტიმალურ ნამრავლს $A_{1..n}$. ამგვარად, ასეთი ოპტიმალური განთავსების ღირებულება არის $A_{1..k}$-ის გამოთვლის ღირებულება, პლუს $A_{k+1..n}$-ის გამოთვლის ღირებულება, პლუს მათი გამრავლების ღირებულება.

რაც უფრო ნაკლები იქნება გამრავლების ოპერაციების რაოდენობა $A_{1..k}$ და $A_{k+1..n}$ ნამრავლების გამოთვლისას, მით უფრო ნაკლები იქნება გამრავლებათა საერთო რაოდენობა. აქედან გამომდინარე, შეგვიძლია დავასკვნათ, რომ მატრიცათა მიმდევრობის გადამრავლების ოპტიმალური ამონახსნი იყენებს ქვეამოცანათა ოპტიმალურ ამონახსნებს. ეს ნიშნავს, რომ შეგვიძლია გამოვიყენოთ დინამიკური პროგრამირების მეთოდი.

რეკურენტული თანაფარდობა. ახლა გამოვსახოთ ოპტიმალური ამონახსნის ღირებულება ქვეამოცანების ოპტიმალური ამონახსნებით. ასეთ ქვეამოცანებს წარმოადგენენ $A_{i..j}$ ნამრავლების გამოთვლისთვის ფრჩხილთა ოპტიმალური განთავსების ამოცანები, სადაც $1 \le i \le j \le n$. აღვნიშნოთ $m[i,j]$-ით ნამრავლთა მინიმალური რაოდენობა, რომელიც საჭიროა $A_{i..j}$-ის გამოსათვლელად. შევნიშნოთ, რომ მთელი $A_{1..n}$ ნამრავლის ღირებულება იქნება $m[1,n]$.

$m[i,j]$ რიცხვები ასე გამოითვლება. თუ $i = j$, მაშინ $m[i,i] = 0$, რადგან მიმდევრობა ერთი მატრიცისაგან შედგება და გამრავლება საჭირო არაა. თუ $i < j$, მაშინ ვისარგებლოთ უკვე განხილული ოპტიმალური ამონახსნის სტრუქტურით. ვთქვათ $m[i,j]$-ის გამოთვლის პროცესში ყველაზე ბოლოს ხდება $A_{i..k}$ და $A_{k+1..j}$ ნამრავლების გადამრავლება, სადაც $i \le k < j$. რადგან $A_{i..k}A_{k+1..j}$-ის გამოსათვლელად საჭიროა $p_{i-1}p_kp_j$ გამრავლების შესრულება, ცხადია, რომ

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

ამ თანაფარდობის მიღებისას ჩვენ ვგულისხმობდით, რომ ჩვენთვის ცნობილია k-ს ოპტიმალური მნიშვნელობა. რადგან იგი წინასწარ ცნობილი ვერ იქნება (მის შესახებ წინასწარ ცნობილი მხოლოდ ისაა, რომ $i \le k < j$ და k-მ შეიძლება მიიღოს მხოლოდ $j - i$ განსხვავებული მნიშვნელობა. მათ შორის ერთ-ერთი ოპტიმალურია და მის საპოვნელად საჭიროა გადავარჩიოთ ეს მნიშვნელობები. მივიღეთ რეკურენტული ფორმულა:

$$m[i,j] = \begin{cases} 0 & \text{როცა } i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{როცა } i < j \end{cases}$$

$m[i,j]$ რიცხვები ქვეამოცანების ოპტიმალური ამოხსნების ღირებულებებია. ეს რეკურენტული თანადობა მისი გამოთვლის საშუალებას გვაძლევს, მაგრამ რადგან ჩვენ გვინდა არა მარტო ოპტიმალური ღირებულება, არამედ მისი მიღწევის გზის ცოდნაც (ანუ ფრჩხილების განთავსების ოპტიმალური რიგი), დაგვჭირდება კიდევ ერთი აღნიშვნა $s[i,j] = k$, რომლისთვისაც $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

ოპტიმალური ღირებულების განსაზღვრა რეკურსიით, დინამიკური პროგრამირების გარეშე. წინა ბიჯზე მიღებული რეკურენტული თანაფარდობის მიხედვით იოლად შეიძლება რეკურსიული ალგორითმის აგება, თუმცა მისი მუშაობის დრო, სრული გადარჩევის მსგავსად, ექსპონენციალურად იქნება დამოკიდებული n-ის მნიშვნელობაზე. დროში ნამდვილ მოგებას მხოლოდ მაშინ ვნახავთ, თუ გამოვიყენებთ იმ ფაქტს, რომ ქვეამოცანების რიცხვი მცირეა და არ აღემატება n^2-ს. რეკურსიული ალგორითმი, რომლის ფსევდოკოდი ქვემოთაა მოყვანილი, მხოლოდ ზემოთ მოყვანილ რეკურენტულ თანადობას ეყრდნობა და ამიტომ უწევს ერთი და იგივე ქვეამოცანის მრავალჯერ

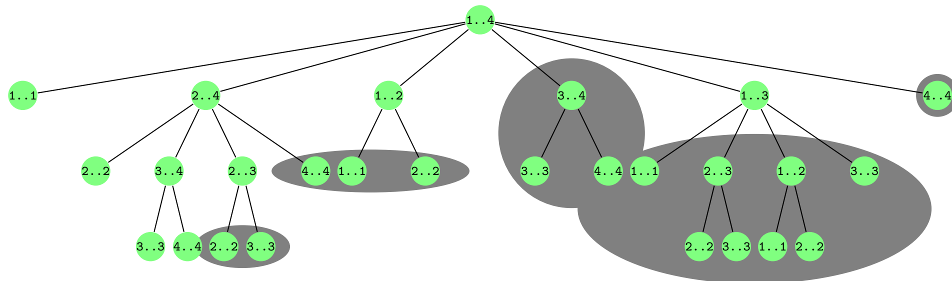
ამოხსნა რეკურსიული ხის სხვადასხვა განშტოებაში, სწორედ ამითაა განპირობებული მისი მუშაობის ექსპონენციალური დრო.

Algorithm 21: Matrix Chain Multiplication (Recursive)

Input: მატრიცათა მიმდევრობის განზომილებები p[0..n], ფუნქციის გამოძახებისას i=1 და j=n
Output: i-დან j-მდე მატრიცათა მიმდევრობის გადამრავლებისათვის საჭირო მინიმალური ოპერაციების რაოდენობა

```
1 REC-MAT-CHAIN(p, i, j) :
2   if i == j :
3     return 0;
4   m[i][j] = ∞;
5   for k=i; k<=j-1; k++ :
6     q = REC-MAT-CHAIN(p,i,k) + REC-MAT-CHAIN(p,k+1,j) + p[i-1]p[k]p[j];
7     if q < m[i][j] :
8       m[i][j] = q;
9   return m[i][j];
```

(ცხადია, ∞-ის ნაცვლად ვწერთ უდიდეს რიცხვს იმ ტიპის რიცხვებს შორის, რომელსაც ეკუთვნის ამ ფუნქციის მნიშვნელობათა სიმრავლე). ნახ. 4-ზე მოცემულია რეკურსიის ხე REC-MAT-CHAIN(p,1,4)-ისათვის. ყოველ წვეროში ჩაწერილია i-ს და j-ს მნიშვნელობები. რუხი ფერით აღნიშნულია ის წვეროები, რომელთა მნიშვნელობების გამოთვლა განმეორებით ხდება. ცხადია, რომ ეს ალგორითმი შორსაა ოპტიმალურისგან, რისი მიზეზიც არის ის, რომ ერთი და იგივე ქვეამოცანა მრავალჯერ იხსნება თავიდან. დინამიკური პროგრამირების მეთოდით შეიძლება ამის თავიდან აცილება.



ოპტიმალური ღირებულების განსაზღვრა დინამიკური პროგრამირების მეთოდით "ქვემოდან ზემოთ". ვნახოთ თუ როგორ შეიძლება გამოვთვალოთ s[i][j] და m[i][j] რიცხვები მეთოდით "ქვემოდან ზემოთ" (ე.ი. დაწყებული უმარტივესიდან, ჯერ ამოვხსნათ ყველაზე მარტივი ქვეამოცანები, შემდეგ კი მათი საშუალებით უფრო რთულები და ა.შ.).
m ფუნქციის მნიშვნელობების გამოთვლისას, ალგორითმი თანმიმდევრულად წყვეტს ამოცანას ფრჩხილების ოპტიმალური განთავსების შესახებ $1, 2, \ldots, n$ თანამამრავლისათვის. ფორმულიდან ჩანს, რომ j-i+1 მატრიცის გადამრავლების ღირებულება - რიცხვი m[i][j], დამოკიდებულია მხოლოდ j-i+1 რიცხვზე ნაკლები მატრიცების გადამრავლების ღირებულებებზე. კერძოდ, $k = i, i+1, \ldots, j-1$ მნიშვნელობებისათვის ვღებულობთ, რომ $A_{i..k}$ არის k-i+1 ¡ j-i+1 მატრიცის ნამრავლი, ხოლო $A_{k+1..j}$ - j-k ¡ j-i+1 მატრიცის ნამრავლი.
თავიდან (სტრ. 2) ვიღებთ m[i][i]=0 $i = 1, \ldots, n$: ერთი მატრიცისგან შემდგარი მიმდევრობის ნამრავლის ღირებულებები ნულის ტოლია. შემდეგ, (სტრ. 3-8) ციკლის პირველი შესრულებისას, გამოითვლება 2 სიგრძის მქონე ქვემიმდევრობების ნამრავლების მინიმალური ღირებულებები m[i][i+1] $i = 1, \ldots, n-1$. შემდეგ გამოითვლება მინიმალური ღირებულებები 3 სიგრძის მქონე ქვემიმდევრობების ნამრავლებისათვის m[i][i+2] $i = 1, \ldots, n-2$ და ა.შ. ყოველ ბიჯზე m[i][j]-ის მნიშვნელობის გამოთვლა დამოკიდებულია მხოლოდ მანამდე გამოთვლილ m[i][k]-სა და m[k+1][j]-ის მნიშვნელობებზე.
ნახ. 5-ზე ნაჩვენებია როგორ მიმდინარეობს გამოთვლები n=6-სათვის.
რადგანაც ჩვენ განვსაზღვრავთ $m[i, j]$-ებს მხოლოდ $i \leq j$-სათვის, გამოიყენება ცხრილის მხოლოდ ის ნაწილი, რომელიც მთავარი დიაგონალის ზემოთაა მოთავსებული. მასივი შებრუნებულია და მთავარი დიაგონალი ჰორიზონტალურადაა. ქვემოთ მითითებულია მატრიცათა თანმიმდევრობა. $m[i, j]$ რიცხვი ანუ $A_i A_{i+1} \ldots A_j$ ნამრავლის მინიმალური ღირებულება - იმყოფება შესაბამისი სტრიქონისა და სვეტის გადაკვეთაზე. ყოველ ჰორიზონტალურ რიგში თავმოყრილია ფიქსირებული სიგრძის მქონე ქვემიმდევრობების ნამრავლთა ღირებულებები. $m[i, j]$ უჯრედის შესავსებად საჭიროა ვიცოდეთ $p_{i-1}p_k p_j$ ნამრავლი $k = i, i+1, \ldots, j-1$-სათვის და $m[i, j]$-ის ქვედა-მარჯვენა და ქვედა-მარცხენა უჯრედების მნიშვნელობები.

Algorithm 22: Matrix Chain Multiplication (DP Bottom-Up)

Input: მატრიცათა მიმდევრობის განზომილებები p[0..n]
Output: m[i][j]-ში i-დან j-მდე მატრიცათა მიმდევრობის გადამრავლებისათვის საჭირო მინიმალური ოპერაციების რაოდენობა, s[i][j]-ში ინფორმაცია ფრჩხილების ოპტიმალური განლაგებისთვის

```
1  MATRIX-CHAIN-ORDER(p) :
2      n = len(p)-1;
3      for i=1; i<=n; i++ :
4          m[i][i] = 0;
5      for r=2; r<=n; r++ :
6          for i=1; i<=n-r+1; i++ :
7              j = i + r - 1;
8              m[i][j] = ∞;
9              for k=i; k<=j-1; k++ :
10                 q = m[i][k] + m[k+1][j] + p[i-1]p[k]p[j];
11                 if q < m[i][j] :
12                     m[i][j] = q;
13                     s[i][j] = k;
14     return m, s;
```

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(a) m მატრიცა

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

(b) s მატრიცა

ნახ. 8.5

ნახ. 5-ზე მოცემულ მატრიცათა ზომებია: $A_1 - 30 \times 35$, $A_2 - 35 \times 15$, $A_3 - 15 \times 5$, $A_4 - 5 \times 10$, $A_5 - 10 \times 20$, $A_6 - 20 \times 25$. m-ის ცხრილში ნაჩვენებია მხოლოდ ის უჯრედები, რომლებიც არ მდებარეობენ მთავარი დიაგონალის ქვემოთ, ხოლო s-ის ცხრილში - უჯრედები, რომლებიც მკაცრად ზემოთ მდებარეობენ. ყველა მატრიცის გადასამრავლებლად საჭირო ნამრავლთა მინიმალური ღირებულებაა m[1][6]=15125. ერთნაირი შეფერილობის მქონე უჯრედთა წყვილები ერთდროულად შედიან m[2][5]-ის გამოსათვლელი ფორმულის მარჯვენა ნაწილში:

$$m[2][5] = \min \begin{cases} m[2][2] + m[3][5] + p[1]p[2]p[5] = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2][3] + m[4][5] + p[1]p[3]p[5] = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2][4] + m[5][5] + p[1]p[4]p[5] = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

ამ ალგორითმის მუშაობის დროა $O(n^3)$, რადგან ჩადგმული ციკლების რაოდენობა 3-ია და თითოეულის მთვლელი არ ღებულობს n-ზე მეტ მნიშვნელობას. ოპტიმალური ღირებულების განსაზღვრა დინამიკური პროგრამირების მეთოდით "ზემოდან ქვემოთ". "ზემოდან ქვემოთ" მომუშავე ალგორითმის შემთხვევაში, ყოველი ამოხსნილი ქვეამოცანის პასუხი უნდა დავიმახსოვროთ სპეციალურ ცხრილში. პირველად შეხვედრილი ქვეამოცანის პასუხი გამოითვლება და შეიტანება ცხრილში. შემდგომში ამ ქვეამოცანის პასუხი აიღება ცხრილიდან. ჩვენს მაგალითში პასუხების ცხრილის შემოღება იოლია, რადგან ქვეამოცანები დანომრილია (i,j) წყვილებით (უფრო რთულ შემთხვევებში შეიძლება ჰეშირების გამოყენება). რეკურსიული ალგორითმების ასეთ გაუმჯობესებას ინგლისურად უწოდებენ memoization.
თავდაპირველად იმის მისათითებლად, რომ ცხრილში ეს უჯრედი შევსებული არ არის, შემდეგ შესაბამისი ქვეამოცანის პირველად ამოხსნისას უჯრედში ჩაიწერება პასუხი და თუ ამ ქვეამოცანის ამოხსნა კიდევ გახდა საჭირო, პასუხი უკვე პირდაპირ ცხრილიდან აიღება. ნახ. 5-ზე ჩანს ის ეკონომია, რომელსაც ასეთი მიდგომა განაპირობებს. რუხად შეფერილი წვეროები შეესაბამება იმ შემთხვევებს, როცა განმეორებითი გამოთვლები საჭირო აღარ არის.
ალგორითმი MEMOIZED-MATRIX-CHAIN საჭიროებს $O(n^3)$ დროს. ცხრილში ელემენტების რაოდენობა n^2-ის რიგისაა. ყოველი პოზიცია ერთხელ ინიციალიზირდება (MEMOIZED-MATRIX-CHAIN(p)-ს მეოთხე სტრიქონი)

Algorithm 23: Matrix Chain Multiplication (DP Top-Down)

Input: მატრიცათა მიმდევრობის განზომილებები p[0..n]
Output: m[i][j]-ში i-დან j-მდე მატრიცათა მიმდევრობის გადამრავლებისათვის საჭირო მინიმალური ოპერაციების რაოდენობა, s[i][j]-ში ინფორმაცია ფრჩხილების ოპტიმალური განლაგებისთვის

```
1  MEMOIZED-MATRIX-CHAIN(p) :
2      n = len(p)-1;
3      for i=1; i<=n; i++ :
4          for j=i; j<=n; j++ :
5              m[i][j] = ∞;
6      return LOOKUP-CHAIN(p, 1, n);
7  LOOKUP-CHAIN(p, i, j) :
8      if m[i][j] < ∞ :
9          return m[i][j];
10     if i == j :
11         m[i][j] = 0;
12     else:
13         for k=i; k<=j-1; k++ :
14             q = LOOKUP-CHAIN(p,i,k) + LOOKUP-CHAIN(p,k+1,j) + p[i-1]p[k]p[j];
15             if q < m[i][j] :
16                 m[i][j] = q;
17                 s[i][j] = k;
18     return m[i][j];
```

ერთადერთხელ ივსება - LOOKUP-CHAIN(p,i,j)-ის პირველი გამოძახებისას მოცემული i,j პარამეტრებით. n^2-ის რიგის პირველი გამოძახებებიდან თითოეული მოითხოვს O(n) დროს (რადგან შიგნით გამოყენებული განმეორებითი გამოძახებები ითხოვს მხოლოდ O(1) დროს, რადგან მათი წაკითხვა ხდება დამახსოვრებული ცხრილიდან), ამიტომ მუშაობის საერთო დრო არის $O(n^3)$.

ოპტიმალური ამონახსნის აგება. ალგორითმი MATRIX-CHAIN-ORDER პოულობს ნამრავლთა მინიმალურ რაოდენობას, რომელიც საჭიროა მატრიცათა მიმდევრობის გადასამრავლებლად. ახლა მოვძებნოთ ფრჩხილების განთავსება, რომელიც მიგვიყვანს ნამრავლთა ასეთ რიცხვამდე. ამისათვის გამოვიყენოთ ცხრილი s[1][n] უჯრედში ჩაწერილია უკანასკნელი გამრავლების ადგილი ფრჩხილების ოპტიმალური განთავსებისას. სხვაგვარად რომ ვთქვათ, $A_{1..n}$-ის ოპტიმალური გზით გამოთვლისას უკანასკნელად შესრულდა $A_{1..s[1,n]}$-ისა და $A_{s[1,n]+1..n}$-ის ნამრავლი. წინა გამრავლებები შეიძლება მოიძებნონ რეკურსიულად. ქვემოთ მოყვანილი რეკურსიული პროცედურა ოპტიმალურად განათავსებს ფრჩხილებს $A_{i..j}$ ნამრავლში, შემდეგი შემავალი მონაცემებით: s ცხრილი, i და j ინდექსები. PRINT-OPTIMAL-PARENS(s,1,n) პროცედურის გამოძახების შემდეგ, ფრჩხილები ოპტიმალურად განთავსდება $A_{1..n}$ მატრიცათა ნამრავლში.

Algorithm 24: Print Optimal Parens

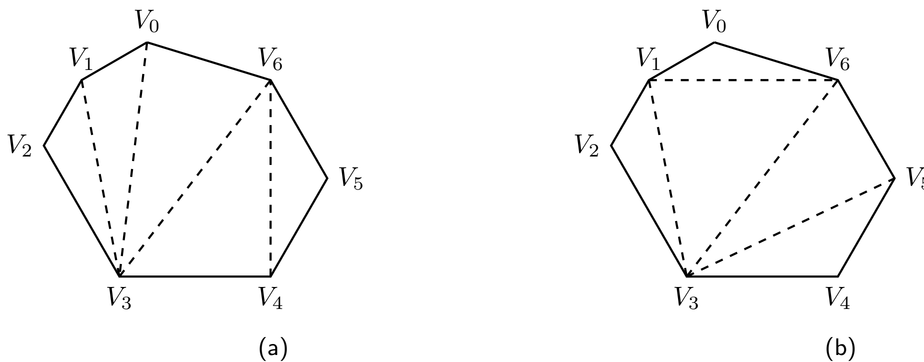
Input: MATRIX-CHAIN-ORDER(p)-თი გამოთვლილი s მატრიცა
Output: ბეჭდავს i-დან j-მდე მატრიცათა მიმდევრობის ოპტიმალური გადამრავლებისათვის საჭირო ფრჩხილების განლაგებას

```
1  PRINT-OPTIMAL-PARENS(s, i, j) :
2      if i == j :
3          print('A_i');
4      else:
5          print('(');
6          PRINT-OPTIMAL-PARENS(s, i, s[i][j]);
7          PRINT-OPTIMAL-PARENS(s, s[i][j]+1, j);
8          print(')');
```

ზემოთ მოყვანილი მაგალითისათვის პროცედურა განალაგებს ფრჩხილებს შემდეგნაირად: $((A_1(A_2A_3))((A_4A_5)A_6))$.

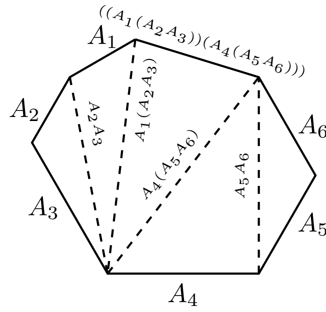
8.6 მრავალკუთხედის ოპტიმალური ტრიანგულაცია

მიუხედავად გეომეტრიული ფორმულირებისა, ეს ამოცანა ძალიან წააგავს მატრიცების გადამრავლების ამოცანას. მრავალკუთხედი (polygon) - ესაა სიბრტყეზე მოთავსებული შეკრული ტეხილი, რომელიც შედგება მრავალკუთხედის გვერდებად (sides) წოდებული მონაკვეთებისაგან. წერტილს, რომელშიც ერთდება ორი მეზობელი გვერდი, წოდებენ წვეროს (vertex). მრავალკუთხედს, რომელიც თავის თავს არ კვეთს, უწოდებენ მარტივს (simple). სიბრტყის წერტილებს, რომლებიც მდებარეობენ მარტივი მრავალკუთხედის შიგნით, უწოდებენ მრავალკუთხედის შიგა არეს (interior), მრავალკუთხედის გვერდების გაერთიანებას უწოდებენ საზღვარს (boundary), ხოლო სიბრტყის ყველა დანარჩენი წერტილების სიმრავლეს - გარეთა არეს (exterior). მარტივ მრავალკუთხედს უწოდებენ ამოზნექილს (convex), თუკი შიგა არეში ან საზღვარზე მდებარე ნებისმიერი ორი წერტილის შემაერთებელი მონაკვეთის არც ერთი წერტილი არა მოთავსებული მრავალკუთხედის გარეთ.
ამოზნექილი მრავალკუთხედი შეგვიძლია აღვწეროთ მისი წვეროების ჩამოთვლით საათის ისრის საწინააღმდეგო მიმართულებით: $P =< v_0, v_1, \ldots, v_{n-1} >$ მრავალკუთხედს აქვს n გვერდი: $v_0v_1, v_1v_2, \ldots, v_{n-1}v_0$. თუ v_i და v_j წვეროები არ წარმოადგენენ მეზობელ წვეროებს მაშინ v_iv_j მონაკვეთს უწოდებენ მრავალკუთხედის დიაგონალს (ცჰორდ). v_iv_j დიაგონალი მრავალკუთხედს ჰყოფს ორად - $< v_i, v_{i+1}, \ldots, v_j >$ და $< v_j, v_{j+1}, \ldots, v_i >$. მრავალკუთხედის ტრიანგულაცია (triangulatuion) - ესაა დიაგონალთა ერთობლიობა, რომლებიც მრავალკუთხედს სამკუთხედებად ჰყოფს. ამ სამკუთხედების გვერდებს წარმოადგენენ საწყისი მრავალკუთხედის გვერდები და ტრიანგულაციის დიაგონალები.



ნახ. 8.6

ამ ნახაზზე ნაჩვენებია მრავალკუთხედის ორგვარი ტრიანგულაცია. ტრიანგულაცია ასევე შეიძლება განისაზღვროს, როგორც იმ დიაგონალთა მაქსიმალური სიმრავლე, რომლებიც არ იკვეთებიან.
n-კუთხედის ნებისმიერი ტრიანგულაციისას საჭიროა სამკუთხედების ერთნაირი რაოდენობა. კერძოდ, ის იყოფა n-2 სამკუთხედად და ამისათვის გამოიყენება n-3 დიაგონალი. მრავალკუთხედის ყველა კუთხის ჯამი ტოლია 180°-ის ნამრავლისა სამკუთხედების რიცხვზე ტრიანგულაციაში.
ოპტიმალური ტრიანგულაციის ამოცანა (optimal triangulation problem) მდგომარეობს შემდეგში: მოცემულია ამოზნექილი მრავალკუთხედი $P =< v_0, v_1, \ldots, v_{n-1} >$ და წონითი ფუნქცია ω, რომელიც განსაზღვრულია ისეთი სამკუთხედების სიმრავლეზე, რომელთა წვეროები მდებარეობს P-ს წვეროებში. საჭიროა მოიძებნოს ტრიანგულაცია, რომლისათვისაც სამკუთხედის წონათა ჯამი უმცირესი იქნება.
წონითი ფუნქციის ყველაზე მარტივი მაგალითია სამკუთხედის ფართობი. მაგრამ ამ შემთხვევაში ნებისმიერი ტრიანგულაციის სამკუთხედების წონათა ჯამი ერთი და იგივეა და არის მრავალკუთხედის ფართობის ტოლი. გაცილებით შინაარსიან მაგალითს წარმოადგენს წონად სამკუთხედის პერიმეტრის განხილვა. ოპტიმალური ტრიანგულაციის ამოცანის ამოხსნის ქვემოთ მოყვანილი ალგორითმი შეიძლება გამოყენებულ იქნას ნებისმიერი სახის წონითი ფუნქციისათვის.
არსებობს კავშირი მრავალკუთხედის ტრიანგულაციასა და ფრჩხილების განლაგებას შორის.
ტრიანგულირებული მრავალკუთხედის ყველა გვერდს ერთის გარდა მივუწეროთ თითო თანამამრავლი. შემდეგ სამკუთხედში, რომლის ორი გვერდი უკვე მონიშნულია, მესამე გვერდს მივუწეროთ ამ ორი გვერდის ნამრავლი. საბოლოოდ, თავდაპირველად მოუნიშნავ გვერდზე მივიღებთ ფრჩხილების სრულ განლაგებას. (იხ. ნახ.)
შევნიშნოთ, რომ მატრიცების გადამრავლების ამოცანა ოპტიმალური ტრიანგულაციის ამოცანის კერძო შემთხვევაა. ვთქვათ, ჩვენ უნდა გამოვთვალოთ $A_1 \times A_2 \times \cdots \times A_n$, სადაც A_i წარმოადგენს $p_{i-1} \times p_i$ მატრიცას. განვიხილოთ $n+1$-კუთხედი $P = v_0, v_1, \ldots, v_n$ და წონითი ფუნქცია, მაშინ ტრიანგულაციის ღირებულება გადამრავლებათა რიცხვის ტოლი იქნება ფრჩხილების შესაბამისი განლაგებისას.
მიუხედავად იმისა, რომ მატრიცების გადამრავლების ამოცანა ოპტიმალური ტრიანგულაციის ამოცანის კერძო შემთხვევაა, ზემოთ განხილული ალგორითმი იოლად შეიძლება გადავაკეთოთ ტრიანგულაციის ამოცანაზე. ამი-



ნახ. 8.7

სათვის საკმარისია სათაურში p შევცვალოთ v-თი, კოდში შევცვალოთ $q = m[i,k] + m[k+1,j] + w(\triangle v_i v_j v_k)$ და ალგორითმის მუშაობის შედეგად m[1][n] გახდება ოპტიმალური ტრიანგულაციის წონის ტოლი.
განვიხილოთ რეკურენტული ფორმულა. ვთქვათ, $m[i,j]$ არის $< v_{i-1}, v_i, \ldots, v_j >$ მრავალკუთხედის ოპტიმალური ტრიანგულაციის წონა, სადაც $1 \leq i \leq j \leq n$. მთელი მრავალკუთხედის ოპტიმალური ტრიანგულაციის წონა ტოლია $m[1,n]$. ჩავთვალოთ, რომ $< v_{i-1}, v_i >$ "ორკუთხედების" წონა არის 0. მაშინ $m[i,i] = 0$, ნებისმიერი $i = 1, 2, \ldots, n$-სათვის. თუ $j - i \geq 1$, მაშინ $< v_{i-1}, v_i, \ldots, v_j >$ მრავალკუთხედში გვაქვს არანაკლებ სამი წვეროსი და ყველა k-სათვის $i \leq k \leq j-1$ შუალედიდან უნდა ვიპოვოთ ასეთი ჯამის მინიმუმი: $\triangle v_{i-1} v_k v_j$-ის წონა პლუს $< v_{i-1}, v_i, \ldots, v_k >$-ს ოპტიმალური ტრიანგულაციის წონა პლუს $< v_{i-1}, v_i, \ldots, v_k >< v_k, v_{k+1}, \ldots, v_j >$-ს ოპტიმალური ტრიანგულაციის წონა.
ამიტომ:

$$m[i,j] = \begin{cases} 0 & \text{როცა } i = j \\ \min_{i \leq k < j} \{min[i,k] + min[k+1,j] + w(\triangle v_{i-1} v_k v_j)\} & \text{როცა } i < j \end{cases}$$

შემდეგი მოქმედებები ანალოგიურია წინა ამოცანის მოქმედებებისა.

8.7 სავარჯიშოები

1. ოპტიმალურად განალაგეთ ფრჩხილები მატრიცათა შემდეგი მიმდევრობის ნამრავლში:
   (ა) $A_1 - 2 \times 3$, $A_2 - 3 \times 4$, $A_3 - 4 \times 1$
   (ბ) $A_1 - 1 \times 2$, $A_2 - 2 \times 3$, $A_3 - 3 \times 1$, $A_4 - 1 \times 4$
2. იპოვეთ შემდეგი მიმდევრობების უდიდესი საერთო ქვემიმდევრობა:
   (ა) $X = (A, R, H, M, K, O)$ $Y = (R, B, H, O, K, M)$
   (ბ) $X = (1, 0, 1, 1, 0, 0)$ $Y = (1, 1, 0, 0, 1, 1)$

თავი 9

ხარბი ალგორითმები

გარკვეული კლასის ოპტიმიზაციის ამოცანების ამოხსნა ხშირად შეიძლება უფრო მარტივად და სწრაფად, ვიდრე ეს კეთდება დინამიკური პროგრამირების მეთოდით. მაგალითად, ე.წ. **ხარბი ალგორითმების** (greedy algorithms) გამოყენებისას, ყოველ ბიჯზე კეთდება ლოკალურად ოპტიმალური არჩევანი იმ იმედით, რომ საბოლოო შედეგიც ოპტიმალური იქნება. ეს მიდგომა ყოველთვის არ ამართლებს, მაგრამ ზოგიერთი ამოცანისათვის მართლაც ძალიან ეფექტურია. რაც მთავარია, არსებობს სტანდარტული პროცედურები იმის გასარკვევად, კორექტულად იმუშავებს თუ არა მოცემული ამოცანისთვის ხარბი ალგორითმი.

9.1 ამოცანა განაცხადების შერჩევაზე

ამოცანის დასმა და შესაბამისი ხარბი ალგორითმი. ვთქვათ, მოცემულია n განაცხადი ერთსა და იმავე აუდიტორიაში მეცადინეობის ჩატარებაზე. ორი განსხვავებული მეცადინეობა არ შეიძლება დროში გადაიფაროს. ყოველ განაცხადში მითითებულია მეცადინეობის დაწყებისა და დამთავრების დრო (i-ური განაცხადისათვის შესაბამისად s_i და f_i). სხვადასხვა განაცხადები შეიძლება გადაიკვეთონ, მაგრამ ამ შემთხვევაში დაკმაყოფილდება მხოლოდ ერთი მათგანი. ჩვენ ვაიგივებთ თითოეულ განაცხადს $[s_i, f_i)$ შუალედთან, ასე რომ ერთი მეცადინეობის დამთავრების დრო შეიძლება დაემთხვეს მეორის დაწყებას და ასეთი სიტუაცია გადაკვეთად არ ითვლება.
ზოგადად i და j ნომრების მქონე განაცხადები თავსებადია (compatible), თუკი $[s_i, f_i)$ და $[s_j, f_j)$ ინტერვალები არ თანაიკვეთებიან (სხვა სიტყვებით, თუ $f_i \leq s_j$ ან $f_j \leq s_i$). **ამოცანა განაცხადების შერჩევაზე** (activity-selection problem) მდგომარეობს იმაში, რომ ამოვარჩიოთ ერთმანეთთან თავსებადი მაქსიმალური რაოდენობის განაცხადი.
ამ ამოცანაში, ხარბი ალგორითმი მუშაობს შემდეგნაირად: დავუშვათ განაცხადები დალაგებულია დამთავრების დროის ზრდადობის მიხედვით:

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

თუკი მონაცემები დალაგებული არ არის, მისი დალაგება შესაძლებელია $O(n \log(n))$ დროში. თუ განაცხადებს ერთნაირი დამთავრების დრო აქვთ, ისინი შეიძლება განლაგდნენ ნებისმიერად.
თუ f-ს და s-ს განვიხილავთ როგორც შესაბამის მასივებს, ალგორითმს ექნება სახე:

Algorithm 25: Greedy Activity Selector

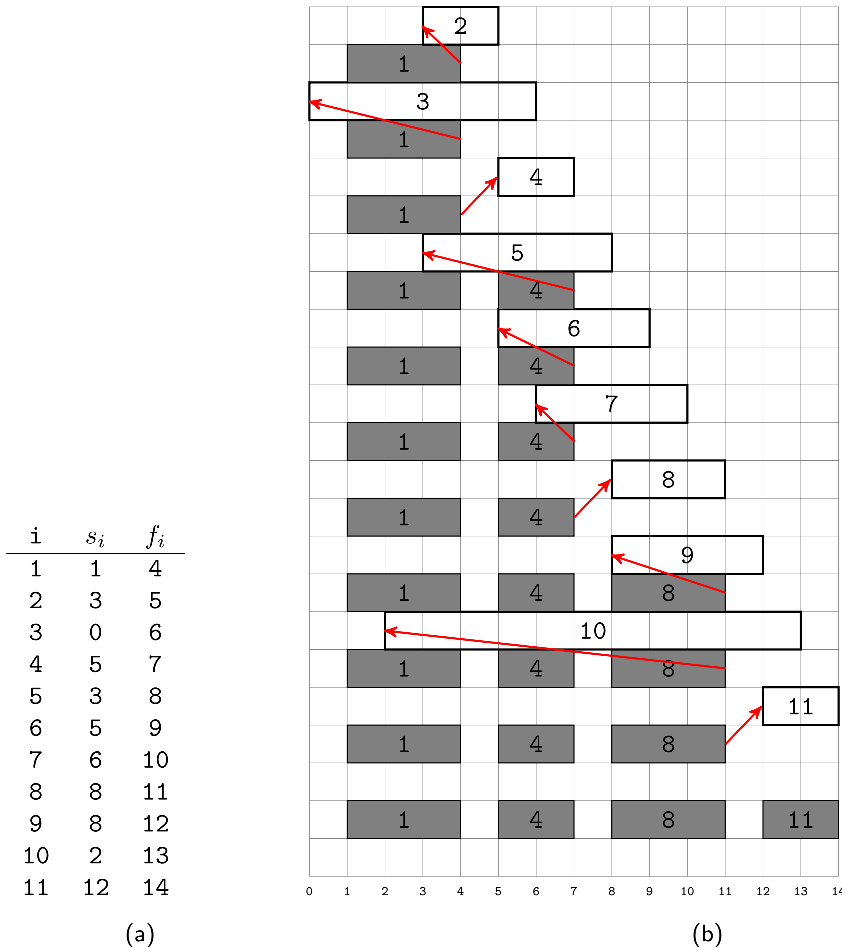
Input: ori masivi, s[i] ganacxadis dawyebis dro, xolo f[i] misi damTavrebis dro
Output: SerCeuli ganacxadebis nomrebi

```
1 GREEDY-ACTIVITY-SELECTOR(s, f) :
2     n = len(s);
3     A = {1};
4     j = 1;
5     for i=2; i<=n; i++ :
6         if s[i] >= f[j] :
7             A = A ∪ {i};
8             j = i;
9     return A;
```

ალგორითმის მუშაობა მოცემულია მომდევნო ნახაზზე. A სიმრავლე შედგება ამორჩეული განაცხადების ნომრებისაგან, ხოლო j უკანასკნელია ამ ნომრებს შორის, ამასთან:

$$f_j = \max\{f_k : k \in A\}$$

რადგან განაცხადები დალაგებულია მათი დამთავრების დროის მიხედვით, თავიდან A სიმრავლე შეიცავს ნომერ 1 განაცხადს და $j = 1$ (2-3 სტრიქონები). შემდეგ (ციკლი 4-7 სტრიქონებში) ვეძებთ განაცხადს, რომელიც არ იწყება j ნომრის მქონე განაცხადის დამთავრებამდე. თუკი ასეთი მოიძებნა, მას ჩავრთავთ A სიმრავლეში და j ცვლადს მივანიჭებთ მის ნომერს (6-7 სტრიქონი).
თუკი სორტირების დროს არ გავითვალისწინებთ, ალგორითმი მუშაობს n-ის პროპორციულ დროში. როგორც ახასიათებთ ხარბ ალგორითმებს, ყოველ ბიჯზე იგი ისეთ არჩევანს აკეთებს, რომ დარჩენილი თავისუფალი დრო იყოს მაქსიმალური (რათა კიდევ "ბევრი" სხვა განაცხადი ჩაეტიოს).



i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

ნახ. 9.1:

ალგორითმის მართებულობა. როგორც აღვნიშნეთ, ხარბი ალგორითმი ყველა ამოცანაში არ იძლევა სწორ პასუხს, მაგრამ ჩვენ ამოცანაში იგი სწორად მუშაობს, რაშიც გვარწმუნებს შემდეგი.

თეორემა 9.1. ალგორითმი GREEDY-ACTIVITY-SELECTOR გვაძლევს თავსებადი განაცხადების მაქსიმალურად შესაძლებელ რაოდენობას.

Proof. როგორც აღვნიშნეთ, განაცხადები დალაგებულია დამთავრების დროის მიხედვით. ამის გამო, ამ ამოცანაში არსებობს ისეთი ოპტიმალური ამონახსნი, რომელიც შეიცავს პირველ განაცხადს. მართლაც, დავუშვათ საწინააღმდეგო და ვთქვათ, პირველი განაცხადი არ შედის განაცხადების რომელიმე ოპტიმალურ სიმრავლეში, მაშინ ჩვენ შეგვიძლია ამ სიმრავლიდან ყველაზე ადრე დამთავრებული განაცხადი შევცვალოთ პირველი განაცხადით და ამით განაცხადების თავსებადობა არ დაირღვევა, რადგან არც ერთი განაცხადი არ მთავრდება პირველ განაცხადზე ადრე, მათ შორის ისიც, რომელიც შევცვალეთ. მაშასადამე, ამ ცვლილებით არ შეიცვლება განაცხადთა საერთო რაოდენობაც და თუკი მოცემული სიმრავლე ოპტიმალური იყო, ოპტიმალური იქნება ის სიმრავლეც, რომელიც პირველ განაცხადს შეიცავს.

ამის შემდეგ განვიხილოთ მხოლოდ ის განცხადებები, რომლებიც თავსებადია პირველ განაცხადთან (ყველა არათავსებადი შეგვიძლია გავაუქმოთ). შედეგად მივიღებთ იგივე ამოცანას, ოღონდ უფრო ნაკლები რაოდენობის განაცხადებისათვის. ინდუქციის მეთოდით ვასკვნით, რომ ყოველ ბიჯზე ხარბი ამორჩევის საშუალებით მივალთ ოპტიმალურ ამონახსნამდე. □

9.2 როდის გამოვიყენოთ ხარბი ალგორითმი?

ზოგადი სქემა იმის გასაგებად, რომელიმე კერძო ამოცანაში გვაძლევს თუ არა ხარბი ალგორითმი ოპტიმალურ ამონახსნს, არ არსებობს, თუმცა შეიძლება გამოიყოს ორი თავისებურება იმ ამოცანებისათვის, რომლებიც ხარბი ალგორითმით იხსნება. ესაა **ხარბი ამორჩევის პრინციპი** და **ოპტიმალური ქვესტრუქტურა**. იტყვიან, რომ ოპტიმიზაციის ამოცანისათვის გამოყენებადია **ხარბი ამორჩევის პრინციპი** (greedy-shoice property), თუკი ლოკალურად ოპტიმალური (ხარბი) არჩევანების მიმდევრობა იძლევა გლობალურად ოპტიმალურ ამონახსნს. განსხვავება ხარბ ალგორითმებსა და დინამიკურ პროგრამირებას შორის იმაში მდგომარეობს, რომ ხარბი ალგორითმი ყოველ ბიჯზე ირჩევს საუკეთესო ვარიანტს და ამის შემდეგ ცდილობს გააკეთოს იგივე დარჩენილ ვარიანტებში, ხოლო დინამიკური პროგრამირების ალგორითმით წინასწარ ხდება შედეგების გამოთვლა ყველა ვარიანტისათვის.
იმის დამტკიცება, რომ ხარბი ალგორითმი ოპტიმალურ ამონახსნს იძლევა, არ წარმოადგენს ტრივიალურ ამოცანას. ტიპურ შემთხვევებში ასეთი მტკიცება ხდება ზემოთ მოყვანილ თეორემაში მოცემული სქემით. თავიდან ვამტკიცებთ, რომ პირველ ბიჯზე ხარბი ამორჩევა არ კეტავს გზას ოპტიმალური ამონახსნისაკენ - ნებისმიერი ამონახსნისათვის არსებობს სხვაც, რომელიც შეთანხმებულია ხარბ ამორჩევასთან და არაა პირველზე უარესი. ამის შემდეგ უნდა ვაჩვენოთ, რომ ქვეამოცანა, რომელიც წარმოიშვა პირველ ბიჯზე ხარბი ამორჩევის შემდეგ, საწყისის ანალოგიურია.
ხარბი ალგორითმებით ამოხსნად ამოცანებს უნდა ჰქონდეთ **ოპტიმალური ქვესტრუქტურა** (optimal substructure): მთელი ამოცანის ოპტიმალური ამონახსნი იგება ქვეამოცანების ოპტიმალური ამონახსნებისგან. ამ თვისებას ჩვენ უკვე გავეცანით დინამიკური პროგრამირების განხილვის დროს. ორივე აღნიშნული მეთოდი - დინამიკური პროგრამირებაც და ხარბი ალგორითმებიც ემყარება ქვეამოცანების ოპტიმალურობის თვისებას, ამიტომ ხშირად იქმნება ერთი მეთოდის ნაცვლად მეორის გამოყენების საფრთხე. თუკი ერთ შემთხვევაში - ხარბი ალგორითმის მაგიერ დინამიკური პროგრამირების გამოყენებისას, მაინც შესაძლებელია სწორი პასუხის მიღება (თუმცა აუცილებლად წავაგებთ დროში), მეორე შემთხვევის დროს (დინამიკურის ნაცვლად ხარბი ალგორითმის გამოყენებისას) პრაქტიკულად შეუძლებელია სწორი პასუხის მიღება. ამ ორი მეთოდის თავისებურება განვიხილოთ ერთი კარგად ცნობილი ოპტიმიზაციის ამოცანის ორ ვარიანტზე.
ზურგჩანთის დისკრეტული ამოცანა (0-1 knapsack problem): ვთქვათ ქურდი შეიპარა საწყობში, რომელშიც ინახება n სხვადასხვა სახის ნივთი, თითო - თითო ეგზემპლარი (ამაზე მიუთითებს 0-1 ამოცანის დასახელებაში: ნივთი ან არის, ან არა). i-ური ნივთი ღირს v_i დოლარი და იწონის w_i კილოგრამს (v_i და w_i - მთელი რიცხვებია). ქურდს სურს წაიღოს მაქსიმალური საფასურის საქონელი, ამასთან მაქსიმალური წონა, რომელიც მან ზურგჩანთით შეიძლება წაიღოს, W-ს ტოლია (W მთელი რიცხვია). რომელი ნივთები უნდა ჩაალაგოს ქურდმა ზურგჩანთაში?
ზურგჩანთის უწყვეტი ამოცანა (fractional knapsack problem): დისკრეტული ამოცანისაგან იმით განსხვავდება, რომ ქურდს შეუძლია დაანაწევროს მოპარული ნივთები და ისინი ზურგჩანთაში ჩაალაგოს ნაწილ-ნაწილ და არა მთლიანად (შეგვიძლია ვთქვათ, რომ დისკრეტულ ამოცანაში ქურდს საქმე აქვს ოქროს ზოდებთან, ხოლო უწყვეტ ამოცანაში - ოქროს ქვიშასთან).
ორივე ამოცანას ზურგჩანთის შესახებ აქვს ოპტიმალურობის თვისება ქვეამოცანებისათვის. მართლაც, დისკრეტული ამოცანის შემთხვევაში თუკი ოპტიმალურად გავსებული ზურგჩანთიდან ამოვიღებთ j ნომრის მქონე ნივთს, მივიღებთ ოპტიმალურ ამონახსნს $W - w_j$ მაქსიმალური წონის მქონე ზურგჩანთისა და n-1 ნივთისათვის (ყველა ნივთი j-ურის გარდა). მსგავსი მსჯელობა მართებულია უწყვეტი ამოცანისთვისაც.

ნახ. 9.2:

მიუხედავად იმისა, რომ ამოცანები ზურგჩანთის შესახებ ძალიან წააგავს ერთმანეთს, ხარბი ალგორითმი პოულობს ოპტიმალურ ამონახსნს უწყვეტი ამოცანისათვის და ვერ პოულობს - დისკრეტულისათვის. უწყვეტი ამოცანისათვის ალგორითმი ასე გამოიყურება. ყველა საქონლისათვის გამოვთვალოთ 1 კილოგრამის ფასი. თავდაპირველად ქურდი აიღებს ყველაზე ძვირფასი საქონლის მაქსიმალურ რაოდენობას, თუკი ზურგჩანთაში ადგილი კიდევ დარჩა აიღებს ფასით მომდევნო საქონელს და ასე გააგრძელებს მანამ, ვიდრე ზურგჩანთა არ შეივსება. ამ ალგორითმის მუშაობის დროა $O(n\log(n))$, რაც განპირობებულია იმით, რომ მონაცემებს წინასწარ სჭირდება სორტირება.
იმაში დასარწმუნებლად, რომ ანალოგიური ხარბი ალგორითმი არ მუშაობს სწორად დისკრეტული ამოცანისათვის, განვიხილოთ ბოლო ნახაზზე მოცემული შემთხვევა. აქ მოცემულია 10, 20 და 30 კგ წონის სამი ნივთი, რომელთა ღირებულება შესაბამისად არის 60$, 100$ და 120$. მასის ერთეულის ღირებულება იქნება 6$, 5$ და 4$. ხარბი სტრატეგიის თანახმად ქურდმა თავიდან პირველი ნივთი უნდა ჩადოს ზურგჩანთაში, რადგან ის ყველაზე ძვირფასია. მაგრამ ცხადია, რომ უფრო მომგებიანია მე-2 და მე-3 ნივთების არჩევა, რადგან ამ შემთხვევაში წაღებული ნივთების საერთო ღირებულება 220$ იქნება, მაშინ როცა ხარბი ალგორითმით აირჩეოდა 160$-ის საქონელი. უწყვეტი ამოცანისათვის ხარბი ალგორითმის მუშაობა ნაჩვენებია ბოლო ნახაზის (გ) ნაწილში.
დისკრეტული ამოცანისთვის ხარბი სტრატეგია არ მუშაობს, თუ ხარბი არჩევანების დამთავრების შემდეგ ჩანთაში კიდევ დარჩება თავისუფლი ადგილი, რაც შეამცირებს მოპარული ნივთების ფასს, გაანგარიშებულს წონის ერთეულზე. იმის გასარკვევად, ჩავდოთ თუ არა მოცემული ნივთი ჩანთაში, საჭიროა ორი ქვეამოცანის ამოხსნა: როცა მოცემული ნივთი აუცილებლად იქნება ჩანთაში და როცა მოცემული ნივთი არ იქნება ჩანთაში. ასე მიიღება ერთმანეთის გადამფარავი ქვეამოცანები, რაც წარმოადგენს ტიპურ სიმპტომს დინამიკური დაპროგრამებისთვის.

9.3 მატროიდები

ხარბ ალგორითმებს უკავშირდება კომბინატორიკის ერთ-ერთი მიმართულება - **მატროიდების თეორია**. იგი ხშირად გამოიყენება იმის საჩვენებლად, რომ ხარბი ალგორითმი იძლევა ოპტიმუმს.
მატროიდი ეწოდება M=(S,I) წყვილს, რომელიც აკმაყოფილებს შემდეგ პირობებს:

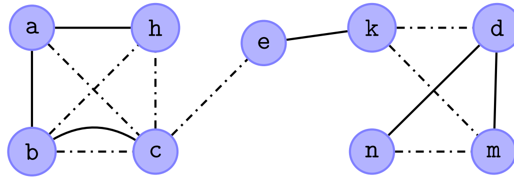
1. S - სასრული არაცარიელი სიმრავლეა
2. I - არაცარიელი ოჯახია S-ის ქვესიმრავლების; I-ში შემავალ ყოველ ქვესიმრავლეს ეწოდება დამოუკიდებელი (independent), და აუცილებლად სრულდება მემკვიდრეობითი (hereditary) თვისება:
$$(B \in I \text{ და } A \subseteq B) \implies A \in I$$
3. თუ $A \in I$, $B \in I$ და $|A| < |B|$, მაშინ არსებობს ისეთი ელემენტი $x \in B\backslash A$, რომ $A \cup \{x\} \in I$. ამას ეწოდება **გადაცვლის** თვისება (exchange property).

განვიხილოთ ორი მაგალითი.
ვთქვათ, S არის რომელიღაც მატრიცის სტრიქონების სიმრავლე, ხოლო რამდენიმე სტრიქონისგან შედგენილ სიმრავლეს ვუწოდოთ დამოუკიდებელი, თუ ეს სტრიქონები წრფივად დამოუკიდებელია ჩვეულებრივი აზრით. ადვილი საჩვენებელია, რომ ასე მიიღება მატროიდი. ეს მაგალითი იმითაა საინტერესო, რომ მატროიდების თეორიაში ეს პირველი მაგალითი იყო და სახელიც (მატროიდი) აქედან მოდის.
ვთქვათ G არის არაორიენტირებული გრაფი. განვმარტოთ (S_G, I_G) წყვილი შემდეგნაირად: S_G წარმოადგენს გრაფის წიბოთა ერთობლიობას, ხოლო I_G შედგება წიბოთა აციკლური ქვესიმრავლეებისგან (წიბოთა აციკლურ ქვესიმრავლეში, არ არსებობს ამ ქვესიმრავლის წიბოებისგან შედგენილი გზა, რომლის საწყისი წვერო ამავდროულად მის ბოლოს წარმოადგენს).
წინასწარ აღვნიშნოთ რამდენიმე ცნობილი ფაქტი გრაფების შესახებ.
ვთქვათ A არის G-ს წიბოების რაიმე ქვესიმრავლე, არა აუცილებლად აციკლური. A-ს წიბოებით გრაფის წვეროთა სიმრავლე V იყოფა რამდენიმე თანაუკვეთი ქვესიმრავლის გაერთიანებად:

$$V = V_1 \cup V_2 \cup \cdots \cup V_k$$
$$V_i \cap V_j = \varnothing \text{ თუ } i \neq j$$

ისე რომ ყოველი V_i არის მაქსიმალური სიმრავლე, რომლის ნებისმიერი ორი წვერო შეერთებულია A-ს წიბოებით შედგენილი გზით. სხვა სიტყვებით, ამ შემთხვევაში ამბობენ რომ $V_1, V_2, \ldots, V_k$ არის **ბმული კომპონენტები**. რადგან წიბოთა სხვადასხვა სიმრავლე განსხვავებულ ბმულ კომპონენტებად ხლეჩენ წვეროთა სიმრავლეს, ამიტომ საჭიროების შემთხვევაში ხაზს ვუსვამთ თუ წიბოთა რომელმა სიმრავლემ შექმნა ბმული კომპონენტების

მოცემული სისტემა და ვამბობთ, მაგალითად, რომ $V_1, V_2, \ldots, V_k$ არის (A) ბმული კომპონენტების სისტემა. მაგალითად, შემდეგ ნახაზზე გამოსახული გრაფის წვეროების სიმრავლე არის $\{a, b, c, d, e, h, k, m, n\}$, მისი წიბოების სიმრავლე გაყოფილია ორად: უწყვეტი მონაკვეთები შეადგენენ A სიმრავლეს რომელიც აციკლურია, ხოლო წყვეტილი მონაკვეთები შეადგენენ B სიმრავლეს. (A) ბმული კომპონენტებია $\{a, b, c, h\}$, $\{e, k\}$, $\{d, m, n\}$, ხოლო (B) ბმული კომპონენტებია: $\{a, b, c, h, e\}$ და $\{d, k, m, n\}$. B-ს წიბოებისგან შედგენილი გზა $cbhc$ ქმნის ციკლს, ამიტომ B სიმრავლე არაა აციკლური. შევნიშნოთ კიდევ ერთი ფაქტი.



ნახ. 9.3:

ლემა 9.1. თუ A სიმრავლე აციკლურია და v_1v_2 არის წიბო, რომლის ერთი წვერო ეკუთვნის რომელიმე (A) - ბმულ კომპონენტს V_1, ხოლო მეორე წიბო სხვა (A) - ბმულ კომპონენტს V_2, მაშინ $v_1v_2 \notin A$ და v_1v_2-ის დამატებით A კვლავ აციკლური რჩება.

Proof. პირველი დასკვნა ცხადია რადგან v_1, v_2 აქამდე არ ერთდებოდა რაიმე გზით და ამიტომ ეკუთვნოდნენ სხვადასხვა ბმულ კომპონენტს. გარდა ამისა, ორი ბმული კომპონენტის გაერთიანება მოგვცემს ბმულ სიმრავლეს $(V_1 \cup V_2)$-ის ნებისმიერი ორი წვერო გახდა შეერთებადი რაიმე გზით) რომლის ნებისმიერი ორი წვერო $A \cup \{v_1v_2\}$ სიმრავლის წიბოებისგან შედგენილი ერთადერთი გზით ერთდება (რადგან გზის ფრაგმენტები ძველ კომპონენტებში ერთადერთია და კომპონენტებს შორის კავშირიც ერთადერთია). □

მაგალითად, როგორც ჩანს ნახაზზე, თუ A სიმრავლეს დავამატებთ ce წიბოს, იგი კვლავ აციკლურია, ოღონდ კომპონენტების რაოდენობა ერთით შემცირდება. აქვე კარგად ჩანს, რომ ნებისმიერი წიბოს დამატება არ ნიშნავს აციკლურობის შემარჩუნებას.

თეორემა 9.2. თუ $G = (V, E)$ წარმოადგენს არაორიენტირებულ გრაფს, მაშინ $M = (S_G, I_G)$ წარმოადგენს მატროიდს.

Proof. წიბოთა აციკლური სიმრავლის ნებისმიერი ქვესიმრავლე აგრეთვე აციკლურია, ამიტომ I_G სიმრავლეს აქვს მემკვიდრეობითი თვისება და დასამტკიცებელი დარჩა, რომ მას აქვს გადაცვლის თვისებაც.
ვთქვათ ახლა, A არის G-ს წიბოების რაიმე აციკლური ქვესიმრავლე. ინდუქციის მეთოდით ვაჩვენოთ, რომ V-ში (A) ბმული კომპონენტების რაოდენობა არის $|V|-|A|$. მართლაც, თუ A ცარიელია, მაშინ $|A| = 0$ და ყოველი წვერო ცალკე კომპონენტია, სულ $|V|$ ცალი. ვთქვათ ეს სამართლიანია $|A| = k$-სთვის და დავამატოთ A-ს ერთი წიბო ისე, რომ იგი კვლავ აციკლური დარჩეს. მაშინ, ახალი წიბო ვერ შეაერთებს ერთი და იგივე ბმული კომპონენტის ორ წვეროს (ისინი უკვე შეერთებულია რაღაც გზით და აციკლურობა დაირღვევა). ამგვარად, ერთი ახალი წიბოს დამატება აციკლურობის შენარჩუნებით იწვევს კომპონენტების რიცხვის ერთით შემცირებას.
ვთქვათ A და B - წიბოთა აციკლური ქვესიმრავლეებია G-ში და —B—¿—A—. (A) ბმული კომპონენტების რაოდენობა არის —V—-—A—, ხოლო (B) ბმული კომპონენტების რაოდენობა არის —V—-—B—. რადგან —V—-—B—¡—V—-—A—, ამიტომ არსებობს B-ს წვეროების მიერ შექმნილი ბმული კომპონენტი T, რომლის ორი წვერო ეკუთვნის A-ს წვეროების მიერ შექმნილ ორ განსხვავებულ ბმულ კომპონენტს, თუ არადა აღმოჩნდება რომ (B) ბმული კომპონენტები ჩალაგებულია (A) ბმულ კომპონენტებში, ანუ მათი რაოდენობა არანაკლებია (რადგან ყოველ (A) ბმულ კომპონენტს აქვს თანაკვეთა რომელიღაც (B) ბმულ კომპონენტთნ). ე.ი. არსებობს B სიმრავლის წიბოებისგან შედგენილი გზა, რომელიც იწყება რომელიღაც (A) ბმულ კომპონენტში V_1 და მთავრდება მის გარეთ. მაშასადამე, აუცილებლად არსებობს B-ს ისეთი (u,v) წიბო, რომ $u \in V_1$ და $v \notin V_1$. ზემოთ დამტკიცებული ლემის ძალით გასაგებია, რომ თუ (u,v) წიბოს დავამატებთ A-ს, კვლავ აციკლურ ქვესიმრავლეს მივიღებთ. □

არაორიენტირებული G გრაფში სხვანაირადაც შეიძლება მატროიდების განმარტება. მაგალითად, შეგვიძლია I_G სიმრავლედ გამოვაცხადოთ წიბოთა ნებისმიერი ქვესიმრავლის ერთობლიობა, ან მხოლოდ ცარიელი სიმრავლე. მაგრამ ესაა ხელოვნური და უშინაარსო მაგალითები.
M=(S,I) მატროიდში $x \notin A$ ელემენტს ეწოდება A-ს **გაფართოება**, თუ $A \cup \{x\} \in I$. მაგალითად, გრაფიკულ მატროიდში წიბო წარმოადგენს წვეროთა დამოუკიდებელი A სიმრავლის გაფართოებას, თუ მისი დამატება არ ქმნის ციკლს.
მატროიდის დამოუკიდებელ ქვესიმრავლეს ეწოდება **მაქსიმალური**, თუ არ არსებობს უფრო დიდი ზომის დამოუკიდებელი ქვესიმრავლე, რომელიც მას მოიცავს.

თეორემა 9.3. მოცემული მატროიდის ყოველი მაქსიმალური დამოუკიდებელი ქვესიმრავლე შედგება ერთი და იგივე რაოდენობის ელემენტებისგან.

Proof. თუ A და B მაქსიმალური დამოუკიდებელი ქვესიმრავლეებია და —A—¡—B—, მაშინ გადაცვლის თვისების ძალით არსებობს ისეთი $x \notin A$, რომ $A \cup \{x\} \in I$. მაგრამ ეს ნიშნავს წინააღმდეგობას მაქსიმალურობასთან. □

მაგალითად განვიხილოთ გრაფიკული მატროიდი M_G, რომელიც შეესაბამება ბმულ G გრაფს. M_G-ს ყოველი მაქსიმალური დამოუკიდებელი ქვესიმრავლე არის ხე (ბმული აციკლური გრაფი) —V—-1 წიბოთი და ერთმანეთთან აერთებს გრაფის ყველა წვეროს. ასეთ ხეს G გრაფის მინიმალური დამფარავი ხე ეწოდება.
M=(S,I) მატროიდს ვუწოდოთ **წონადი**, თუ S სიმრავლეზე განსაზღვრულია (წონის) ფუნქცია მნიშვნელობებით დადებით რიცხვებში. ამ ფუნქციის გავრცელება შეგვიძლია S-ის ქვესიმრავლეებზე: ქვესიმრავლის წონა განისაზღვრება როგორც მისი ელემენტების წონათა ჯამი:

$$w(A) = \sum_{x \in A} w(x)$$

მაგალითად, გრაფიკულ მატრიდში წიბოს წონად შეგვიძლია მისი სიგრძე მივიღოთ, ხოლო ქვეგრაფის წონად - მისი წიბოების სიგრძეთა ჯამი.

9.4 ხარბი ალგორითმები აწონილი მატროიდისთვის

ხშირად, ოპტიმიზაციის ამოცანის ამოხსნა ხარბი ალგორითმით შეიძლება განხილული იქნას როგორც აწონილ M=(S,I) მატროიდში უდიდესი წონის მქონე დამოუკიდებლი ქვესიმრავლის მოძებნის ამოცანა. უდიდესი წონის მქონე დამოუკიდებელ ქვესიმრავლეს ეწოდება აწონილი მატროიდის ოპტიმალური ქვესიმრავლე. მაგალითად, ამ ტიპის ამოცანას წარმოადგენს მინიმალური დამფარავი ხის განსაზღვრის ამოცანა, რომელსაც შემდეგში განვიხილავთ დაწვრილებით.
მოვიყვანოთ ხარბი ალგორითმის ფსევდოკოდი, რომელიც ნებისმიერ აწონილ მატროიდში იძლევა ოპტიმალური ქვესიმრავლის განსაზღვრის საშუალებას; ამ კოდში, M=(S,I) მატროიდისთვის ვგულისხმობთ, რომ S=S(M), I=I(M); წონით ფუნქციას აღნიშნავს w.

Algorithm 26: Greedy

Input: $M = (S, I)$ მატროიდი და წონითი ფუნქცია w
Output:

```
1 GREEDY(M, w) :
2     A = {∅};
3     sort(S(M)); // წონების კლების მიხედვით
4     for ∀x ∈ S(M) :
5         if A ∪ {x} ∈ I(M) :
6             A = A ∪ {x};
7     return A;
```

ალგორითმი მუშაობს შემდეგნაირად. თავიდან $A = \varnothing$, და შევნიშნოთ რომ ცარიელი სიმრავლე დამოუკიდებელია მემკვიდრეობითი თვისების ძალით. შემდეგ ვასორტირებთ S(M)-ის ელემენტებს კლებადობით და პირველიდან დაწყებული, თუ მორიგი ელემენტის დამატება შეიძლება დამოუკიდებლობის დაურღვევლად, ვამატებთ მას. გასაგებია, რომ ბოლოს მიიღება დამოუკიდებელი სიმრავლე. ქვემოთ ვაჩვენებთ, რომ მას აგრეთვე ექნება მაქსიმალური წონა დამოუკიდებელ ქვესიმრავლეთა შორის, მაგრამ ჯერ შევაფასოთ GREEDY(M,w) მუშაობის დრო. სორტირებას მიაქვს $O(n \log(n))$ დრო, სადაც $n = |S|$. სიმრავლის დამოუკიდებლობის შემოწმება ხდება n-ჯერ, რასაც მიაქვს $f(n)$ დრო. მაშინ, ალგორითმის მუშაობის დროა $O(n \log(n) + f(n))$.
ახლა ვაჩვენოთ, რომ ეს ალგორითმი მართლაც იძლევა ოპტიმალური პასუხს.

ლემა 9.2. (ხარბი ამორჩევის თვისება მატროიდისთვის). ვთქვათ, M=(S,I) არის წონადი მატროიდი წონის w ფუნქციით. ვთქვათ, $x \in S$ არის უდიდესი ზომის მქონე ელემენტი ერთელემენტიან დამოუკიდებელ ელემენტებს შორის. მაშინ, x შედის რომელიღაც ოპტიმალურ $A \subseteq S$ ქვესიმრავლეში.

Proof. ვთქვათ, რომელიმე B არის ოპტიმალური ქვესიმრავლე. ვიგულისხმოთ $x \notin B$, თუ არა და დასამტკიცებელი არაფერია.

ავიღოთ $A' = \{x\}$. ეს დამოუკიდებელი სიმრავლეა. გამოვიყენოთ $(|B|-1)$-ჯერ გადაცვლის თვისება, თანდათან ვაფართოებთ A'-ს და ბოლოს ვიღებთ $\{x\}$-ისა B სიმრავლის (—B—-1) ცალი ელემენტის გაერთიანებისგან შედგენილ A სიმრავლეს. ამგვარად, —A—=—B— (ე.ი. A მაქსიმალურია) და w(A)=w(B)-w(y)+w(x), სადაც y არის ერთადერთი ელემენტი B-დან, რომელიც არ ეკუთვნის A-ს. რადგან მემკვიდრეობითი თვისების ძალით B-ს ყოველი ელემენტი დამოუკიდებელია, ამიტომ $w(x) \geq w(y)$, x-ის არჩევის თანახმად. მაშასადამე, $w(A) \geq w(B)$ და A აგრეთვე ოპტიმალურია.
ალგორითმი GREEDY(M,w) სწორ პასუხს იძლევა, როცა დამოუკიდებელი ელემენტი საერთოდ არაა ან მხოლოდ ერთია; დამტკიცებული ლემა გვაძლევს ამოცანის ზომის შემცირების საშუალებას, რადგან დამოუკიდებელი ერთელემენტიანი ელემენტების სიმრავლე - ის არჩევის შემდეგ ერთით შემცირდა. ახლა, თუ ვაჩვენებთ რომ ამოცანის ზომის შემცირების შემდეგ იგივე ტიპის ამოცანა გვრჩება ამოსახსნელი (ე.ი. უფრო მცირე ზომის აწონილ მატროიდში ოპტიმალური ქვესიმრავლის განსაზღვრა), მაშინ ინდუქციის პრინციპის თანახმად დამტკიცებული იქნება ალგორითმ GREEDY(M,w)-ის კორექტულობა. სხვა სიტყვებით, საჩვენებელი დარჩა, რომ ამ ამოცანას აქვს ოპტიმალური ქვესტრუქტურა. □

ლემა 9.3. (ოპტიმალური ქვესტრუქტურის თვისება მატროიდებისთვის). ვთქვათ, M=(S,I) არის წონადი მატროიდი და $x \in S$ არის ისეთი ელემენტი, რომ $\{x\}$ დამოუკიდებელია. მაშინ უდიდესი წონის მქონე დამოუკიდებელი სიმრავლე, რომელიც შეიცავს x-ს, წარმოადგენს გაერთიანებას $\{x\}$-ისა და $M' = (S', I')$ მონოიდის უდიდესი წონის მქონე დამოუკიდებელი სიმრავლისა, სადაც:

$$S' = \{y \in S : \{x,y\} \in S\}$$

$$I' = \{B' \subseteq S \backslash \{x\} : B \cup \{x\} \in I\}$$

ხოლო წონის ფუნქცია წარმოადგენს M მატროიდის წონის ფუნქციის შეზღუდვით S'-ზე.

Proof. განმარტების თანახმად, S-ის ის დამოუკიდებელი სიმრავლეები, რომლებიც შეიცავენ x-ს, მიიღებიან x-ის დამატებით S'-ის დამოუკიდებელ სიმრავლეებზე. ამასთან, მათი წონები განსხვავდება ზუსტად w(x)-ით, ანუ ოპტიმალურ სიმრავლეებს შეესაბამება ოპტიმალურები. □

9.4.1 განრიგის შედგენის ამოცანა

მატროიდების თეორიის გამოყენებით გამოვიკვლიოთ განრიგის შედგენის ამოცანა იმ პირობებში, რომ შეკვეთები ტოლი ხანგრძლივობისაა (შესრულების დროის მიხედვით), შემსრულებლი ერთადერთია, მოცემულია შეკვეთების შესრულების ვადები და გათვალისწინებულია ჯარიმები ამ ვადების დარღვევისთვის.
უფრო კონკრეტულად, მოცემულია შეკვეთებისგან შედგენილი სიმრავლე S, მათგან თითოეულის შესრულებას სჭირდება დროის ზუსტად ერთი ერთეული. S-ის **განრიგი** (schedule) ეწოდება S-ის ელემენტების ისეთ გადანაცვლებას, რომელიც განსაზღვრავს შეკვეთების შესრულების თანმიმდევრობას: პირველი შეკვეთის შესრულება დაიწყება დროის 0 მომენტში და დასრულდება 1 მომენტში, მეორე შეკვეთის შესრულება დაიწყება დროის 1 მომენტში და დასრულდება 2 მომენტში, და ა.შ.
ამ ამოცანაში, შემავალ მონაცემებს წარმოადგენენ:

- $S = \{1, 2, \ldots, n\}$ სიმრავლე, რომლის ელემენტებს ვუწოდებთ შეკვეთებს
- მთელი არაუარყოფითი რიცხვისგან შედგენილი მიმდევრობა $d_1, d_2, \ldots, d_n$, რომლებსაც **შესრულების ვადები** (deadlines) ეწოდებათ ($1 \leq d_i \leq n$ ყოველი i-სთვის, d_i ეკუთვნის i-ურ შეკვეთას)
- n მთელი არაუარყოფითი რიცხვისგან შედგენილი მიმდევრობა $w_1, w_2, \ldots, w_n$, რომლებსაც **ჯარიმები** (penalties) ეწოდებათ (თუ i-ურ შეკვეთა ვერ შესრულდება d_i დროისთვის, გათვალისწინებულია ჯარიმა w_i)

ჩვენი ამოცანაა ისე განვსაზღვროთ ელემენტების თანმიმდევრობა S-ში, რომ ჯარიმების ჯამი იყოს მინიმალური.
რომელიმე შერჩეულ განრიგში, i-ურ შეკვეთას ეწოდება **ვადაგადაცილებული** (late) თუ მისი შესრულება მთავრდება d_i მომენტის შემდეგ, ხოლო წინააღმდეგ შემთხვევაში ეწოდება **დროულად შესრულებული** (early).
ყოველი განრიგის მოდიფიცირება შეიძლება ჯარიმების ჯამის შეუცვლელად ისე, რომ მასში ყველა ვადაგადაცილებული შეკვეთა იდგეს დროულად შესრულებული შეკვეთების შემდეგ. მართლაც, თუ განრიგით გათვალისწინებულია ვადაგადაცილებული y შეკვეთის შესრულება დროულად შესრულებული x შეკვეთის შემდეგ, მაშინ მათი ადგილების შეცვლა არც ჯარიმების ჯამს ცვლის და არც მათ სტატუსს.
უფრო მეტიც, ჯარიმების ჯამის შეუცვლელად შეგვიძლია ყოველ განრიგს მივცეთ **კანონიკური სახე** (canonical form), რომელშიც ვადაგადაცილებული შეკვეთები დგას დროულად შესრულებული შეკვეთების შემდეგ, ხოლო დროულად შესრულებული შეკვეთების შესრულების ვადები დალაგებულია ზრდადობის მიხედვით. მართლაც,

უკვე ვანახეთ რომ შეგვიძლია ვადაგადაცილებული შეკვეთები დავაყენოთ დროულად შესრულებული i შეკვეთების შემდეგ. ვთქვათ ახლა, დროულად შესრულებული i და შეკვეთები სრულდებიან დროის k და k+1 მომენტებში, შესაბამისად, მაგრამ $d_i > d_j$. ვნახოთ თუ რა მოხდება მათთვის ადგილების გაცვლის შემთხვევაში. j-ურ შეკვეთას არავითარი პრობლემა არა აქვს, რადგან კიდევ უფრო ადრე დასრულდება. რადგან $d_i > d_j \geq k+1$, ამიტომ ისიც დროულად დასრულდება, ჯარიმის გარეშე. რადგან ასეთი შესაძლო გადანაცვლებების მთლიანი რიცხვი სასრულია, ამიტომ შესაძლებელია განრიგის მიყვანა კანონიკურ სახეზე.
ამგვარად, განრიგის შედგენის ამოცანა დაიყვანება ისეთი A სიმრავლის განსაზღვრაზე, რომელიც შედგება დროულად შესრულებული შეკვეთებისგან: როგორც კი ეს სიმრავლე მოიძებნება, მთლიანი განრიგის შესადგენად საკმარისია A-ში შემავალი შეკვეთები განვალაგოთ შესრულების ვადების ზრდის მიხედვით, ხოლო დანარჩენი შეკვეთები მათ შემდეგ ნებისმიერი თანმიმდევრობით.
$A \subseteq S$ სიმრავლეს ვუწოდოთ **დამოუკიდებელი**, თუ ამ სიმრავლის შეკვეთებისთვის შესაძლებელია ისეთი განრიგის შედგენა, რომ ყველა შეკვეთა დროულად შესრულდეს. აღვნიშნოთ I-თი ყველა დამოუკიდებელი ქვესიმრავლის ერთობლიობა.
ჩვენი მსჯელობა რომ კონსტრუქციული იყოს, უნდა მივუთითოთ კრიტერიუმი, რომელიც დაგვეხმარება იმის გარკვევაში, არის თუ არა მოცემული სიმრავლე დამოუკიდებელი. ყოველი $t = 1, 2, \ldots, n$-ისთვის აღვნიშნოთ $N_t(A)$-თი A სიმრავლიდან იმ შეკვეთების რაოდენობა, რომელთა შესრულების ვადა არ აღემატება t-ს.

ლემა 9.4. ყოველი $A \subseteq S$ ქვესიმრავლისთვის შემდეგი სამი პირობა არის ერთმანეთის ეკვივალენტური:

1. A არის დამოუკიდებელი სიმრავლე
2. ყოველი $t = 1, 2, \ldots, n$-ისთვის სრულდება $N_t(A) \leq t$
3. თუ A სიმრავლის შეკვეთებს განვალაგებთ შესრულების ვადების ზრდის მიხედვით, მაშინ ყველა შეკვეთა შესრულდება დროულად

Proof. თუ $N_t(A) > t$ რომელიმე t-სთვის, მაშინ დროის პირველ t ინტერვალში შესასრულებელი შეკვეთების რაოდენობა მეტია t-ზე და ამიტომ ერთი მათგანი მაინც აღმოჩნდება ვადაგადაცილებული. ამიტომ 1) $\Longrightarrow$ 2). ვთქვათ, სრულდება 2) და i_k არის იმ შეკვეთის ნომერი, რომლის შესრულების ვადა არის k-ური, თუ ვადებს დავახარისხებთ ზრდადობის მიხედვით. მაშინ, 2)-ის თანახმად, $d_{i_k} \geq k$, ანუ თუ შეკვეთებს განვალაგებთ შესრულების ვადების ზრდის მიხედვით, ყველა მათგანი დროულად შესრულდება. ბოლოს, 3) $\Longrightarrow$ 1) ცხადია. □

საწყისი ამოცანა, ანუ ვადაგადაცილებული შეკვეთების გამო მიღებული ჯარიმების ჯამის მინიმიზაცია - იგივეა რაც არგადახდილი ჯარიმების ჯამის მაქსიმიზაცია, ანუ იმ ჯარიმებისა, რაც უკავშირდება დროულად შესრულებულ შეკვეთებს. შემდეგი თეორემა გვიჩვენებს, რომ ასეთი ოპტიმიზაციის ამოცანა იხსნება ხარბი ალგორითმით.

თეორემა 9.4. ვთქვათ, S არის ტოლი ხანგრძლივობის შეკვეთების სიმრავლე მოცემული შესრულების ვადებით, ხოლო I არის შეკვეთების დამოუკიდებელი ქვესიმრავლლების ერთობლიობა. მაშინ (S, I) წყვილი წარმოადგენს მატროიდს.

Proof. ცხადია, რომ დამოუკიდებელი სიმრავლის ყოველი ქვესიმრავლე ასევე დამოუკიდებელია და დასამტკიცებელი გვრჩება, რომ სრულდება გადაცვლის თვისებაც. ვთქვათ A და B დამოუკიდებელი სიმრავლეებია და —B—¿—A—. შევადაროთ რიცხვები $N_t(B)$ და $N_t(A)$ სხვადასხვა t-სთვის. როცა $t = n$, პირველი რიცხვია მეტი; ვამცირებთ რა t-ს, რაღაც მომენტში ისინი ერთმანეთის ტოლი ხდება და ამ მომენტს ვუწოდოთ k (თუ ეს სულ ბოლოს მოხდება, ვიღებთ $k = 0$). ამგვარად, $N_k(A) = N_k(B)$ თუ $k = 0$ და $N_{k+1}(B) > N_{k+1}(A)$. ამიტომ არსებობს ერთი შეკვეთა მაინც $x \in B \backslash A$, რომლის შესრულების დრო არ აღემატება $(k+1)$-ს. ავიღოთ $A' = A \cup \{x\}$. თუ $t \leq k$, მაშინ $N_t(A') = N_t(A) \leq t$ A სიმრავლის დამოუკიდებლობის ძალით; თუ $t > k$, მაშინ $N_t(A') = N_t(A) + 1 \leq N_t(B) \leq t$ უკვე B სიმრავლის დამოუკიდებლობის ძალით. მაშასადამე, A' დამოუკიდებელია დამტკიცებული ლემის ძალით და (S, I) წყვილისთის სრულდება გადაცვლის თვისება. □

როგორც ვხედავთ, შეკვეთების ოპტიმალური A სიმრავლის განსაზღვრისთვის შეგვიძლია ხარბი ალგორითმის გამოყენება, ხოლო შემდეგ უნდა შევადგინოთ განრიგი, რომელიც ჯერ A სიმრავლის შეკვეთებს განალაგებს შესრულების ვადების ზრდის მიხედვით, ხოლო შემდეგ დანარჩენ შეკვეთებს. ეს არის განრიგის შედგენის ამოცანის ამოხსნა. თუ გამოვიყენებთ ალგორითმს GREEDY, მაშინ მუშაობის დრო იქნება $O(n^2)$, რადგან ალგორითმის მუშაობის პროცესში საჭიროა სიმრავლის დამოუკიდებლობის შემოწმება n-ჯერ, და თითოეულ ასეთ შემოწმებას სჭირდება O(n) ოპერაცია.
შემდეგ ცხრილში მოყვანილია განრიგის შედგენის ამოცანის ერთი ნიმუში.
ხარბი ალგორითმი არჩევს შეკვეთებს 1,2,3,4 შემდეგ იწუნებს შეკვეთებს 5,6 და კვლავ არჩევს 7-ს. შემდეგ შერჩეული შეკვეთები სორტირდება შესრულების დროის მიხედვით და შედგენილი ოპტიმალური განრიგი მიიღებს სახეს: ¡2,4,1,3,7,5,6¿ ჯარიმების ჯამი არის $w_5 + w_6 = 50$.

შეკვეთა	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

ნახ. 9.4:

9.5 სავარჯიშოები

თავი 10

ქვესტრიქონების ძებნის ამოცანა

10.1 აღნიშვნები და ტერმინოლოგია

ვთქვათ, მოცემული გვაქვს Σ ანბანი. მისი ელემენტებისგან შედგენილ სასრული სიგრძის სიმრავლეს ვუწოდოთ **სტრიქონი** (string). Σ^*-ით ავღნიშნოთ Σ ანბანით შედგენილი სტრიქონების სიმრავლე. სასრული სიგრძის **ცარიელი სტრიქონი** (empty string), რომელიც აღინიშნება ϵ-ით, აგრეთვე, ეკუთვნის Σ^*-ს. x სტრიქონის სიგრძე ავღინიშნოთ —x—-ით. x და y სტრიქონების **კონკატენაცია** (concatenation), ავღინიშნოთ xy-ით, მისი სიგრძეა —x—+—y— და შედგება მიმდევრობით x და y სტრიქონების სიმბოლოებისაგან.
ω სტრიქონს ეწოდება x სტრიქონის **პრეფიქსი** (prefix), (აღინიშნება $\omega \sqsubset x$) თუ არსებობს იმავე ანბანზე განსაზღვრული y სტრიქონი, ისეთი, რომ $x = \omega y$. (მაგ.: $x = abcca$, $\omega = ab$, $ab \sqsubset abcca$).
ω სტრიქონს ეწოდება x სტრიქონის **სუფიქსი** (suffix), (აღინიშნება $\omega \sqsupset x$) თუ არსებობს იმავე ანბანზე განსაზღვრული y სტრიქონი, ისეთი, რომ $x = y\omega$. (მაგ.: $x = abcca$, $\omega = cca$, $cca \sqsupset abcca$).
ცარიელი სტრიქონი ϵ, არის ნებისმიერი სტრიქონის პრეფიქსიც და სუფიქსიც. თუ ω არის x-ის პრეფიქსი ან სუფიქსი, მაშინ $|\omega| \leq |x|$. ნებისმიერი x და y სტრიქონებისთვის და ნებისმიერი a სიმბოლოსთვის, $x \sqsupset y$ სრულდება მაშინ და მხოლოდ მაშინ, როცა $xa \sqsupset ya$. ადგილი აქვს შემდეგ ლემას:

ლემა 10.1. (ორი სუფიქსის შესახებ). ვთქვათ, x, y და z სტრიქონებია, რომლებისთვისაც $x \sqsupset z$, $y \sqsupset z$:

- თუ $|x| \leq |y|$ მაშინ $x \sqsupset y$
- თუ $|x| \geq |y|$ მაშინ $y \sqsupset x$
- თუ $|x| = |y|$ მაშინ $x = y$

თუ სტრიქონის პრეფიქსი ან სუფიქსი არ ემთხვევა სტრიქონს, მას უწოდებენ საკუთრივ პრეფიქსს, ან საკუთრივ სუფიქსს.
n სიგრძის T სტრიქონი ავღნიშნოთ T[1..n]-ით. T[1..n]-ის k-სიმბოლოიანი პრეფიქსი ავღნიშნოთ T_k-თი. ამრიგად, $T_0 = \epsilon$ და $T_n = T = T[1..n]$.

10.2 ქვესტრიქონების ძებნის ამოცანის დასმა

ვთქვათ, მოცემული გვაქვს Σ ანბანზე განსაზღვრული n სიგრძის სტრიქონი, რომელსაც ვუწოდებთ ტექსტს და ავღნიშნავთ T[1..n]-ით და m სიგრძის სტრიქონი, რომელსაც ვუწოდებთ ნიმუშს და ავღნიშნავთ P[1..m]-ით (pattern) ($m \leq n$).
ვიტყვით, რომ T ტექსტში P ნიმუში **შედის** s **წანაცვლებით** (occurs with shift s) ან რაც იგივეა, P ნიმუში T ტექსტში **გვხვდება** s+1 **პოზიციიდან** (occurs beginning at position s+1) თუ $0 \leq s \leq n - m$ და T[s+1..s+m]=P[1..m].
თუ P ნიმუში T ტექსტში შედის s წანაცვლებით, მაშინ s-ს **დასაშვებ წანაცვლებას** (valid shift) უწოდებენ, წინააღმდეგ შემთხვევაში - s **დაუშვებელი წანაცვლებაა** (invalid shift).
ქვესტრიქონების ძებნის ამოცანა (string matching problem) მდგომარეობს შემდეგში: ვიპოვოთ ყველა ის დასაშვები წანაცვლება, რომლითაც P ნიმუში შედის T ტექსტში. ქვესტრიქონების ძებნის ამოცანა შეიძლება ასეც ჩამოვაყალიბოთ: ვიპოვოთ ყველა ის s წანაცვლება $0 \leq s \leq n - m$ ინტერვალში, რომლისთვისაც $P \sqsupset T_{s+m}$.

ჩვენ განვიხილავთ ქვესტრიქონების ძებნის რამდენიმე ალგორითმს. ალგორითმები მოცემული იქნება ფსევდოკოდის სახით. ყველა მათგანში **ქვესტრიქონების ძებნის ამოცანაში** ვეძებთ პირველ დასაშვებ წანაცვლებას, რომელიც შეესაბამება მარცხნიდან პირველ ქვესტრიქონს ტექსტში. თუ საჭიროა ყველა ქვესტრიქონის პოვნა ტექსტში, ალგორითმი გააგრძელებს მუშაობას ტექსტის ბოლომდე.
ფსევდოკოდში იგულისხმება, რომ ერთნაირი სიგრძის ორი სტრიქონის შედარება პრიმიტიული ოპერაციაა. სტრიქონების შედარებისას, გარკვეული რაოდენობა სიმბოლოების დამთხვევის შემდეგ, პირველივე არდამთხვევისთანავე პროცესი წყდება. ითვლება, რომ ამ პროცესზე დახარჯული დრო გამოიხატება წრფივი ფუნქციით, რომელიც დამოკიდებულია სტრიქონების ტოლი (დამთხვეული) სიმბოლოების რაოდენობაზე. უფრო ზუსტად, ითვლება, რომ ტესტი "x = y" სრულდება $\Theta(t+1)$ დროში, სადაც t არის ყველაზე გრძელი z სტრიქონის სიგრძე, რომელიც არის ერთდროულად x-ის და y-ის პრეფიქსი ($z \sqsubset x$, $z \sqsubset y$) (რომ გავითვალისწინოთ t=0 შემთხვევა, $\Theta(t)$-ს ნაცვლად ვწერთ $\Theta(t+1)$-ს. ამ სიტუაციაში არ ემთხვევა სტრიქონების პირველივე სიმბოლოები, მაგრამ ამის შესამოწმებლადაც საჭიროა რაღაც დადებითი დრო).

10.3 ქვესტრიქონების ძებნის უმარტივესი ალგორითმი

ქვესტრიქონების ძებნის ყველაზე მარტივ ალგორითმში, რომელიც დამყარებულია "უხეში ძალის" მეთოდზე, s დასაშვები წანაცვლების პოვნა ხდება s-ის ყველა შესაძლო n-m+1 მნიშვნელობისთვის P[1..m]=T[s+1..s+m] პირობის თანმიმდევრობით შემოწმებით.
ალგორითმი მუშაობს შემდეგნაირად: ნიმუშის და ტექსტის პირველ სიმბოლოებს (ელემენტებს) ვუსწორებთ ერთმანეთს და ვადარებთ შესაბამის წყვილებს მარცხნიდან მარჯვნივ, სანამ ყველა m წყვილი არ დაემთხვევა ერთმანეთს (ამ შემთხვევაში, ალგორითმი ასრულებს მუშაობას). თუ შედარებისას აღმოჩნდა განსხვავებული წყვილი, ნიმუშს ვამოძრავებთ ერთი პოზიციით მარჯვნივ და სიმბოლოების შედარება გრძელდება ნიმუშის პირველი სიმბოლოსა და ტექსტის შესაბამისი სიმბოლოს შედარებიდან. შევნიშნოთ, რომ ტექსტის ბოლო პოზიცია, რომელიც შეიძლება ქვესტრიქონის პირველი სიმბოლო იყოს, არის n-m+1.

Algorithm 27: Naive String Matcher

Input: ტექსტი T და ტექსტში საძებნი ნიმუში P
Output: ტექსტში ნიმუშის წანაცვლებები

```
1 NAIVE-STRING-MATCHER(T, P) :
2     n = len(T);
3     m = len(P);
4     for s=0; i<=n-m; s++ :
5         j = 1;
6         while j <= m and P[j] == T[s+j] :
7             j++;
8         if j == m+1 :
9             print(s);
```

ალგორითმის მუშაობის დრო. ჩავთვალოთ შედარების ოპერაცია ძირითად ოპერაციად. შემავალი მონაცემები განისაზღვრება ტექსტის და ნიმუშის სიმბოლოების რაოდენობით. ალგორითმის მუშაობის დრო იქნება:

$$\sum_{s=0}^{n-m}\sum_{j=1}^{m} 1 = \sum_{s=0}^{n-m}(m-1+1) = \sum_{s=0}^{n-m} m = m(n-m+1) \in \Theta(mn)$$

განვიხილოთ მაგალითი, რომელიც შეესაბამება ე.წ. უარეს შემთხვევას, როცა გვიხდება m სიმბოლოს შემცველი ნიმუშის ყველა სიმბოლოს შედარება ტექსტის შესაბამის სიმბოლოებთან, ანუ, როცა პირველი m-1 სიმბოლო ემთხვევა ტექსტის შესაბამის სიმბოლოებს, და ბოლო-არა. სულ დაგვჭირდება (n-m+1)m შედარება. (ჩვენს შემთხვევაში - 9).
მაგალითი 1. ვთქვათ, T=aaaab, P=aab, n=5, m=3.
ამრიგად, უარეს შემთხვევაში, ალგორითმის მუშაობის დროა $\Theta(mn)$, მაგრამ ტიპობრივ შემთხვევებში, მოსალოდნელია, რომ წანაცვლებების უმეტესობა შესრულდება შედარებების მცირე რაოდენობის ჩატარების შემდეგ. ალგორითმის მუშაობის დრო საშუალო შემთხვევაში არსებითად უკეთესია მუშაობის დროზე უარეს შემთხვევაში, კერძოდ, შეიძლება ვაჩვენოთ, რომ იგი ტოლია $(m+n) = \Theta(n)$.
მაგალითი 2. განვიხილოთ ლათინური ანბანის ასოებისგან და ხაზგასმის სიმბოლოებისგან შედგენილ ტექსტში: BESS-KNEW-ABOUT-BAOBABS ნიმუშის - BAOBAB ძებნის ამოცანა უმარტივესი ალგორითმით.

a	a	a	a	b	
a	a	b			3 შედარება
	a	a	b		3 შედარება
		a	a	b	3 შედარება

ტაბულა 10.1: სულ 9 შედარება

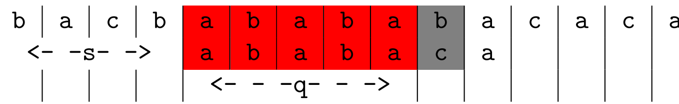
B	E	S	S	-	K	N	E	W	-	A	B	O	U	T	-	B	A	O	B	A	B	S
B	A	O	B	A	B																	
	B	A	O	B	A	B																
		B	A	O	B	A	B															
			B	A	O	B	A	B														
				B	A	O	B	A	B													
					B	A	O	B	A	B												
						B	A	O	B	A	B											
							B	A	O	B	A	B										
								B	A	O	B	A	B									
									B	A	O	B	A	B								
										B	A	O	B	A	B							
											B	A	O	B	A	B						
												B	A	O	B	A	B					
													B	A	O	B	A	B				
														B	A	O	B	A	B			
															B	A	O	B	A	B		
																B	A	O	B	A	B	

ტაბულა 10.2: სულ 24 შედარება

უმარტივესი ალგორითმის არაეფექტურობას განაპირობებს ის, რომ ინფორმაცია ტექსტის შესახებ s წანაცვლების შემოწმების დროს, საერთოდ არ გამოიყენება, მომდევნო წანაცვლების შემოწმებისას.
უმარტივესი ალგორითმისგან განსხვავებით, უფრო სწრაფი ალგორითმები ემყარება ნიმუშის წინასწარი "დამუშავების" იდეას: ნიმუშზე გარკვეული ინფორმაციის მიღებას, მის შენახვას ცხრილში და შემდეგ, ამ ინფორმაციის გამოყენებას ტექსტში ნიმუშის რეალური ძებნისას. ამ იდეას ემყარება ორი ყველაზე ცნობილი ალგორითმი: **ბოიერ-მურის**, **ბოიერ-მურ-ჰორსპულის** (ბოიერ-მურის გამარტივებული შემთხვევა), **კნუტ-მორის-პრატის** ალგორითმები. ბოიერ-მურის, ბოიერ-მურ-ჰორსპულის ალგორითმებში ნიმუშის განხილვა ხდება მარჯვნიდან მარცხნივ, ხოლო კნუტ-მორის-პრატის ალგორითმში - მარცხნიდან მარჯვნივ.

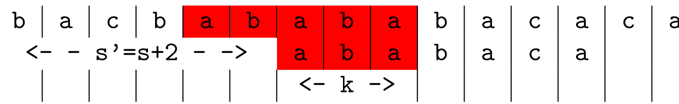
10.4 კნუტ-მორის-პრატის ალგორითმი

ეს ალგორითმი დამოუკიდებლად შექმნეს ერთი მხრივ, კნუტმა (Knuth) და მორისმა (Morris) და მეორე მხრივ, პრატმა(Pratt) 1977 წელს. მისი მუშაობის სისწრაფე განპირობებულია იმით, რომ მოცემული P[1..m] ნიმუშისთვის წინასწარ ვითვლით ე.წ. $\pi[1..m]$ პრეფიქს-ფუნქციას (prefix-function). ალგორითმი მუშაობს შემდეგნაირად: ნიმუშის და ტექსტის პირველ სიმბოლოებს ვუსწორებთ ერთმანეთს და ვადარებთ შესაბამის წყვილებს მარცხნიდან მარჯვნივ, სანამ ყველა m წყვილი არ დაემთხვევა ერთმანეთს (ამ შემთხვევაში, ალგორითმი ასრულებს მუშაობას). თუ შედარებისას აღმოჩნდა განსხვავებული წყვილი, ნიმუშს ვამოძრავებთ მარჯვნივ. ცხადია, ალგორითმი მით ეფექტურია, რაც მეტია წანაცვლების სიდიდე ყოველ ბიჯზე. ამ ალგორითმში წანაცვლების სიდიდე დგინდება ე.წ. $\pi[1..m]$ პრეფიქს-ფუნქციის (prefix-function) გამოყენებით, რომელიც გამოითვლება O(m) დროში. იგი ნიმუშის ელემენტებზეა განსაზღვრული და შეიცავს ინფორმაციას იმის შესახებ, თუ რამდენად ემთხვევა ნიმუში თავის თავს წანაცვლების შემდეგ, რაც საშუალებას გვაძლევს ავიცილოთ თავიდან ზედმეტი შედარებები.
პრეფიქს-ფუნქციის გამოთვლის საჭიროება შეიძლება განვიხილოთ შემდეგ მაგალითზე: ვთქვათ, მოცემულია ლათინური ანბანი Σ, ნიმუში P=ababaca და ტექსტი T=bacbababaabacaca.
ვთქვათ s წანაცვლებისთვის აღმოჩნდა, რომ ნიმუშის პირველი q სიმბოლო (ამ შემთხვევაში q=5) დაემთხვა ტექსტის შესაბამის სიმბოლოებს, ხოლო მომდევნო (მეექვსე) სიმბოლო განსხვავებულია.
ეს ნიშნავს, რომ ჩვენ ვიცით ტექსტის q რაოდენობა T[s+1],...,T[s+q] სიმბოლოები და ის, რომ ნიმუში s წანაცვლებით ტექსტში არ შედის. მოცემული ინფორმაციის საფუძველზე შეიძლება დავასკვნათ, რომ ზოგიერთი



ტაბულა 10.3

წანაცვლება უეჭველად დაუშვებელია. ჩვენს მაგალითში დაუშვებელია წანაცვლება s+1, რადგან ამ დროს ნიმუშის პირველი ელემენტი a აღმოჩნდება ტექსტის s+2-ე ელემენტის - b-ს ქვეშ. s+2 წანაცვლებისას, კი ნიმუშის პირველი 3 ელემენტი დაემთხვევა ტექსტის T[s+3], T[s+4], T[s+5] ელემენტებს (ანუ ტექსტის ჩვენთვის ცნობილ ბოლო 3 ელემენტს) ამიტომ, ამ წანაცვლების შეუმოწმებლად უარყოფა არ შეიძლება.



ტაბულა 10.4

ინფორმაციას დასაშვები წანაცვლებების შესახებ გვაძლევს π[1..m] პრეფიქს-ფუნქცია, რომელიც გამოითვლება ალგორითმის დაწყებამდე და შეიტანება ცხრილში. თუ s წანაცვლებისას, პირველი q სიმბოლო ემთხვევა, მაშინ შემდეგი წანაცვლება, რომელიც შეიძლება იყოს დასაშვები, ტოლია s' = s + (q - π[q]).
საკითხი შეიძლება დავსვათ შემდეგნაირად: ვთქვათ, ნიმუშის P[1..q] სიმბოლოები დაემთხვა ტექსტის T[s+1..s+q] სიმბოლოებს. რას უდრის ის უმცირესი წანაცვლება s'¿s, რომლისთვისაც:

$$P[1..k] = T[s' + 1..s' + k] \tag{10.1}$$

სადაც s'+k = s+q. საუკეთესო შემთხვევაში s' = s+q და s+1,...,s+q-1 წანაცვლებების განხილვა არ დაგვჭირდება. მაგრამ, თუ ნიმუში ისეთია, რომ მისი წანაცვლებისას საკუთარი თავის მიმართ, მისი გარკვეული ელემენტები ემთხვევიან ერთმანეთს, მაშინ s+1,...,s+q-1 წანაცვლებებიდან შეიძლება რომელიმე იყოს დასაშვები. ნებისმიერ შემთხვევაში, s' წანაცვლების განხილვისას, შეგვიძლია არ შევადაროთ ნიმუშის პირველი k სიმბოლო ტექსტის შესაბამის სიმბოლოებს, რადგანაც ისინი აუცილებლად დაემთხვევიან ერთმანეთს. (P[1..k] = T[s'+1..s'+k]-ს საფუძველზე).
s'-ს საპოვნელად, საკმარისია ვიცოდეთ ნიმუში P და რიცხვი q. სახელდობრ, P_q სტრიქონის k სიმბოლოიანი სუფიქსი, რომელიც ემთხვევა ტექსტის T[s'+1..s'+k] სიმბოლოებს. k რიცხვი (10.1) ფორმულაში წარმოადგენს ისეთ უდიდეს რიცხვს, რომლისთვისაც P_k არის P_q-ს სუფიქსი.
P[1..m] ნიმუშის **პრეფიქს-ფუნქცია** ეწოდება ფუნქციას $\pi : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m-1\}$, რომელიც განსაზღვრულია შემდეგნაირად: $\pi[q] = \max\{k : k < q$ და $P_k \sqsupset P_q\}$
სხვა სიტყვებით რომ ვთქვათ, π[q] არის P ნიმუშის იმ უდიდესი პრეფიქსის სიგრძე, რომელიც P_q-ს საკუთრივ სუფიქსს წარმოადგენს.

Algorithm 28: Compute Prefix Function

Input: ტექსტში საძებნი ნიმუში P
Output: პრეფიქს ფუნცქციის მნიშვნელობები მასივში

```
1  COMPUTE-PREFIX-FUNCTION(P) :
2      m = len(P);
3      π[1] = 0;
4      k = 0;
5      for q=2; q<=m; q++ :
6          while k>0 and P[k+1] != P[q] :
7              k = π[k];
8          if P[k+1] == P[q] :
9              k++;
10         π[q] = k;
11     return π;
```

ქვემოთ მოყვანილი კნუტ-მორის-პრატის ალგორითმი ფსევდოკოდის სახით, სადაც ხდება COMPUTE-PREFIX-FUNCTION-ის გამოძახება.

Algorithm 29: ატცჰერ

Input: ტექსტი T და ტექსტში საძებნი ნიმუში P
Output: ტექსტში ნიმუშის წანაცვლებები

```
1  KMP-MATCHER(T, P) :
2      n = len(T);
3      m = len(P);
4      π = COMPUTE-PREFIX-FUNCTION(P);
5      q = 0;
6      for i=1; i<=n; i++ :
7          while q > 0 and P[q+1] != T[i] :
8              q = π[q];
9          if P[q+1] == T[i] :
10             q++;
11         if q == m :
12             print(' ნიმუში აღმოჩენილია წანაცვლებით ', i-m);
13             q = π[q];
```

ალგორითმის მუშაობის დრო. საამორტიზაციო ანალიზის პოტენციალთა მეთოდის გამოყენებით, შეიძლება ვაჩვენოთ, რომ COMPUTE-PREFIX-FUNCTION-ის მუშაობის დრო არის $\Theta(m)$. k (ტექსტისა და ნიმუშის ტოლი სიმბოლოების რაოდენობა) სიდიდის პოტენციალი უკავშირდება მის მიმდინარე მნიშვნელობას ალგორითმში. მე-4 სტრიქონიდან ჩანს, რომ k-ს საწყისი მნიშვნელობაა 0. მე-7 სტრიქონში k მცირდება მისი ყოველი გამოთვლისას, რადგან π[k]¡k. მაგრამ, რადგან $\pi[k] \geq 0$, k არასოდეს არ ხდება უარყოფითი. ერთადერთი სტრიქონი ალგორითმში, რომელშიც შეიძლება შეიცვალოს k-ს მნიშვნელობა, არის მე-8 სტრიქონი, სადაც ის იზრდება არაუმეტეს 1-ით. რადგან ციკლში შესვლამდე k¡q, და q-ს მნიშვნელობა იზრდება ციკლის ყოველი იტერაციისას, k¡q ნარჩუნდება, ისევე როგორც უტოლობა π[q]¡q. while ციკლის ტანის ყოველი შესრულება მე-6 სტრიქონში შეიძლება "გადავიხადოთ" პოტენციური ფუნქციის შემცირებით, რადგანაც π[k]¡k. მე-9 სტრიქონში პოტენციური ფუნქცია იზრდება არაუმეტეს 1-ით, ამიტომ 6-10 სტრიქონებში ციკლის ტანის საამორტიზაციო ღირებულება ტოლია O(1)-ს. რადგან გარე ციკლში იტერაციების რაოდენობა არის $\Theta(m)$ და რადგან პოტენციური ფუნქციის საბოლოო მნიშვნელობა საწყის მნიშვნელობაზე ნაკლები არაა, COMPUTE-PREFIX-FUNCTION-ს მუშაობის ფაქტიური დრო უარეს შემთხვევაში ტოლია $\Theta(m)$-ს.

ანალოგიურად მტკიცდება, რომ KMP-MATCHER-ს მუშაობის დრო არის $\Theta(n)$. ამრიგად, ალგორითმის მუშაობის დრო იქნება $\Theta(n+m)$.

ქვემოთ მოყვანილია პრეფიქს-ფუნქციის ცხრილები ნიმუშებისთვის:

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3	0	1

(a)

i	1	2	3	4	5	6
P[i]	B	A	O	B	A	B
π[i]	0	0	0	1	2	1

(b)

ტაბულა 10.5

მაგალითი 3: განვიხილოთ ლათინური ანბანის ასოებისგან და ხაზგასმის სიმბოლოებისგან შედგენილ ტექსტში: BESS-KNEW-ABOUT-BAOBABS ნიმუშის - BAOBAB ძებნის ამოცანა კნუტ-მორის პრატის ალგორითმით.

B	E	S	S	-	K	N	E	W	-	A	B	O	U	T	-	B	A	O	B	A	B	S
B	A	O	B	A	B																	
	B	A	O	B	A	B																
		B	A	O	B	A	B															
			B	A	O	B	A	B														
				B	A	O	B	A	B													
					B	A	O	B	A	B												
						B	A	O	B	A	B											
							B	A	O	B	A	B										
								B	A	O	B	A	B									
									B	A	O	B	A	B								
										B	A	O	B	A	B							
											B	A	O	B	A	B						
												B	A	O	B	A	B					
													B	A	O	B	A	B				
														B	A	O	B	A	B			
															B	A	O	B	A	B		
																B	A	O	B	A	B	

ტაბულა 10.6: სულ 24 შედარება

მაგალითი 4: განვიხილოთ ლათინური ანბანის ასოებისგან და ხაზგასმის სიმბოლოებისგან შედგენილ ტექსტში: bacbababababacaca ნიმუშის - ababaca ძებნის ამოცანა.

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
a	b	a	b	a	c	a								
	a	b	a	b	a	c	a							
		a	b	a	b	a	c	a						
			a	b	a	b	a	c	a					
				a	b	a	b	a	c	a				
						a	b	a	b	a	c	a		

ტაბულა 10.7: სულ 15 შედარება

10.5 ბოიერ-მურ-ჰორსპულის ალგორითმი

ბოიერ-მურ-ჰორსპულის (შემდეგში, ბ-მ-ჰ-ის) ალგორითმი, ისევე როგორც კ-მ-პ-ის ალგორითმი, ნიმუშის წინასწარი დამუშავების ხარჯზე, ამცირებს ტექსტის და ნიმუშის შედარებების რაოდენობას.
ალგორითმი მუშაობს შემდეგნაირად: ნიმუშის და ტექსტის პირველ სიმბოლოებს ვუსწორებთ ერთმანეთს, ვადარებთ ნიმუშის უკიდურეს მარჯვენა სიმბოლოს ტექსტის შესაბამის სიმბოლოსთან და ვმოძრაობთ მარჯვნიდან მარცხნივ, სანამ ყველა m წყვილი არ დაემთხვევა ერთმანეთს. (ამ შემთხვევაში, საძიებელი ქვესტრიქონი ნაპოვნია და ალგორითმი დასრულებულია) თუ შედარებისას აღმოჩნდა განსხვავებული წყვილი, ნიმუშს ვამოძრავებთ მარჯვნივ. ცხადია, გადაწევა ჯობს, რაც შეიძლება მეტი პოზიციით, მაგრამ არა ქვესტრიქონის გამოტოვების რისკის ხარჯზე. ბ-მ-ჰ-ის ალგორითმი ამ პოზიციების რაოდენობას განსაზღვრავს იმ c სიმბოლოს განხილვით, რომელიც ტექსტის და ნიმუშის ერთმანეთის მიმართ გასწორების მომენტში, მდებარეობს ნიმუშის ბოლო ელემენტის გასწვრივ. გვაქვს ორი შემთხვევა:

1. თუ c სიმბოლო არ ემთხვევა ნიმუშის პირველ m-1 სიმბოლოს, მაშინ ნიმუშის წანაცვლება მარჯვნივ ხდება m პოზიციით.

B	E	S	S	-	K	N	E	W	-	A	B	O	U	T	-	B	A	O	B	A	B	S
B	A	O	B	A	B																	
						B	A	O	B	A	B											

ტაბულა 10.8

(ამ შემთხვევაში K სიმბოლო არ გვხვდება ნიმუშის პირველ 5 სიმბოლოს შორის, რაც ნიშნავს, რომ ნიმუშის წანაცვლებით მარჯვნივ 6 პოზიციით არ გამოვტოვებთ დასაშვებ წანაცვლებას)

2. თუ c სიმბოლო გვხვდება ნიმუშის პირველ m-1 სიმბოლოებს შორის, მაშინ წანაცვლება ხდება ისე, რომ ნიმუშის c-ს ტოლი უკიდურესი მარჯვენა სიმბოლო გაუსწორდეს ტექსტის c სიმბოლოს.

B	E	S	S	-	K	N	E	W	-	A	B	O	U	T	-	B	A	O	B	A	B	S
						B	A	O	B	A	B											
								B	A	O	B	A	B									

ტაბულა 10.9

(ამ შემთხვევაში B სიმბოლო ორჯერ გვხვდება ნიმუშის პირველ 5 სიმბოლოს შორის, ნიმუშის წანაცვლება მარჯვნივ ხდება ისე, რომ ნიმუშის პირველ 5 სიმბოლოს შორის ყველაზე მარჯვნივ მდგომი B სიმბოლო გაუსწორდეს c სიმბოლოს, რომ არ გამოვტოვოთ დასაშვები წანაცვლება)

ამ მაგალითებიდან ჩანს, რომ სიმბოლოების შედარებას მარჯვნიდან მარცხნივ მივყავართ უფრო დიდ წანაცვლებებთან, ვიდრე უმარტივეს ალგორითმში, მაგრამ თუ c სიმბოლოს შევადარებთ ნიმუშის ყოველ ელემენტს, მაშინ შედარებების რაოდენობა იმავე რიგის იქნება, რაც უმარტივეს ალგორითმში. შემავალი მონაცემების "გაუმჯობესების" იდეა ამცირებს ამ შედარებების რაოდენობას. ტექსტის ყოველი ელემენტისათვის (ნიმუშის გათვალისწინებით) წინასწარ ვითვლით წანაცვლების სიდიდეს, რომელიც შემდეგნაირად გამოითვლება:

$$Table[c] = \begin{cases} m & \text{თუ c არ ემთხვევა ნიმუშის პირველ } m-1 \text{ ელემენტს} \\ \text{მანძილი ნიმუშის პირველ } m-1 \text{ ელემენტებს შორის უკიდურესად მარჯვენა } c \text{ ელემენტსა და ბოლო ელემენტს შორის} & \text{წინააღმდეგ შემთხვევაში} \end{cases}$$

წანაცვლების სიდიდეს ვინახავთ ცხრილში.

Algorithm 30: Boyer Moore Horspool Matcher

Input: ტექსტი T და ტექსტში საძებნი ნიმუში P
Output: ტექსტში ნიმუშის წანაცვლებები

```
1  SHIFT-TABLE(T, P) :
2      n = len(T);
3      m = len(P);
4      for k=1; k<=n; k++ :
5          Table[T[k]] = m;
6      for s=1; k<=m-1; s++ :
7          Table[P[s]] = m-s;
8      return Table;
9  BOYER-MOORE-HORSPOOL-MATCHER(T, P) :
10     n = len(T);
11     m = len(P);
12     Table = SHIFT-TABLE(T,P);
13     i = m;
14     while i <= n :
15         k = 0;
16         while k < m and P[m-k] == T[i-k] :
17             k++;
18         if k == m :
19             print(' ნიმუში აღმოჩენილია წანაცვლებით ', i-m+1);
20         i += Table[T[i]];
```

ალგორითმის მუშაობის დრო. წინასწარი დამუშავების ფაზის (წანაცვლების ცხრილი) მუშაობის დრო არის $\Theta(n)$, ხოლო ბოიერ-მურ-ჰორსპულის ალგორითმის მუშაობის დრო უარეს შემთხვევაში არის $\Theta(mn)$.
განვიხილოთ ლათინური ანბანის ასოებისგან და ხაზგასმის სიმბოლოებისგან შედგენილ ტექსტში: BESS-KNEW-ABOUT-BAOBABS ნიმუშის - BAOBAB-ს ძებნის ამოცანა.

c სიმბოლო	A	B	E	S	K	N	W	O	U	T	-
$Table[c]$ წანაცვლება	1	2	6	6	6	6	6	3	6	6	6

(a) წანაცვლებები

B	E	S	S	-	K	N	E	W	-	A	B	O	U	T	-	B	A	O	B	A	B	S
B	A	O	B	A	B																	
						B	A	O	B	A	B											
								B	A	O	B	A	B									
														B	A	O	B	A	B			
																B	A	O	B	A	B	

(b) სულ 13 შედარება

ტაბულა 10.10

10.6 ბოიერ-მურის ალგორითმი

ბოიერ-მურის ალგორითმის ღირსება მდგომარეობს იმაში, რომ ნიმუშის წინასწარი დამუშავების ხარჯზე, მცირდება ნიმუშისა და ტექსტის სიმბოლოების შედარებების რაოდენობა: ზოგიერთი შედარებისთვის წინასწარ ცნობილი ხდება, რომ იგი უსარგებლო იქნება.
ბოიერ-მურის ალგორითმით ქვესტრიქონების ძებნა, ისევე როგორც ბოიერ-მურ-ჰორსპულის ალგორითმში, იწყება ტექსტისა და ნიმუშის პირველი ელემენტების ერთმანეთთან გასწორებით. ტექსტისა და ნიმუშის ელემენტების შედარებას ვიწყებთ ნიმუშის ბოლო ელემენტიდან და ვმოძრაობთ მარჯვნიდან მარცხნივ, სანამ ყველა m წყვილი არ დაემთხვევა ერთმანეთს.(ამ შემთხვევაში, საძიებელი ქვესტრიქონი ნაპოვნია და ალგორითმი დასრულებულია). ვთქვათ, ტექსტისა და ნიმუშის შესაბამისი წყვილების k ($0 \leq k < m$) რაოდენობა დაემთხვა ერთმანეთს, ხოლო k+1-ე წყვილი განსხვავებულია. ამ შემთხვევაში ბოიერ-მურის ალგორითმი განსაზღვრავს წანაცვლების სიდიდეს ორი ევრისტიკის გამოყენებით. ესენია " **სდექ-სიმბოლოს ევრისტიკა**" და " **უსაფრთხო სიმბოლოს ევრისტიკა**". თითოეული მათგანი იძლევა მნიშვნელობას, რომელთა შორისაც უდიდესის არჩევით, შეიძლება გამოვითვალოთ ისეთი მაქსიმალური წანაცვლება, რომ არ გამოვტოვოთ დასაშვები.
წანაცვლების პირველი სიდიდე - **სდექ-სიმბოლოს წანაცვლება** (bad-symbol shift) განისაზღვრება ტექსტის იმ c ელემენტით (დავარქვათ მას სდექ-სიმბოლო) რომელიც შედარებისას პირველი არ დაემთხვა ნიმუშის შესაბამის ელემენტს (ეს ელემენტი შეიძლება იყოს პირველივე ელემენტი. ამ შემთხვევაში k=0).
თუ c ელემენტი არ შედის ნიმუშში, მაშინ ნიმუში უნდა გადავაადგილოთ მარჯვნივ ისე, რომ სდექ-სიმბოლო აღმოჩნდეს წანაცვლების გარეთ. წანაცვლება გამოითვლება ფორმულით: Table[c]-k, სადაც Table[c] წანაცვლების ცხრილის ელემენტია, ხოლო k ტექსტისა და ნიმუშის ტოლი ელემენტების რაოდენობა:

B	E	S	S	-	K	N	E	W	-	A	B	O	U	T	-	B	A	O	B	A	B	S
						B	A	O	B	A	B											
											B	A	O	B	A	B						

ტაბულა 10.11

ერთმანეთს დაემთხვა ნიმუშის და ტექსტის ბოლო ორი AB ელემენტი k=2, სდექ-სიმბოლოა "-". ნიმუში გადაადგილდება მარჯვნივ Table[-]-k, ანუ 6-2=4 პოზიციით. იხილეთ წანაცვლების ცხრილი 10.11.
თუ c ელემენტი შედის ნიმუშში და Table[c]-k¿0, მაშინ წანაცვლება ისევ ამ ფორმულით გამოითვლება, ანუ ხდება ნიმუშის გადაადგილება Table[c]-k პოზიციით.

B	E	S	S	-	K	N	E	W	-	A	B	O	U	T	-	B	A	O	B	A	B	S
					B	A	O	B	A	B												
						B	A	O	B	A	B											

ტაბულა 10.12

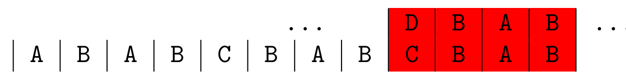
არ დაემთხვა ერთმანეთს ნიმუშის ბოლო ელემენტი B და ტექსტის შესაბამისი ელემენტი A (სდექ-სიმბოლო), ნიმუში გადაადგილდება მარჯვნივ Table[A]-k, ანუ 1-0=1 პოზიციით.

თუ სდექ-სიმბოლო c შედის ნიმუშში მაგრამ Table[c]-k¡0, მაშინ, ცხადია, ვერ წავანაცვლებთ ნიმუშს უარყოფითი რაოდენობა პოზიციით. ამ შემთხვევაში, ვიყენებთ "უხეში ძალის" პრინციპს და ვამოძრავებთ ნიმუშს ერთი პოზიციით მარჯვნივ.

ამრიგად, "სდექ-სიმბოლოს ევრისტიკით" წანაცვლების სიდიდე d_1 ტოლია Table[c]-k, თუ ეს სიდიდე დადებითია, და უდრის 1-ს, თუ იგი არადადებითია.

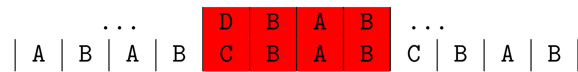
$$d_1 = \max\{Table[c] - k, 1\}$$

წანაცვლების მეორე სიდიდე - **საერთო სუფიქსის წანაცვლება** (good-suffix shift) განისაზღვრება ნიმუშის k ელემენტის დამთხვევით ტექსტის შესაბამის ელემენტებთან და იგი ემყარება შემდეგ იდეას: ვთქვათ, ნიმუშში გვხვდება სიმბოლოების განლაგება, რომელიც ემთხვევა ნიმუშის k - სიმბოლოიან სუფიქსს, მაშინ ნიმუში შეიძლება წავანაცვლოთ იმდენი პოზიციით, რომ k - სიმბოლოიანი სუფიქსის ტოლი სიმბოლოების განლაგება, რომელიც მარჯვნიდან მეორეა, დაემთხვეს k - სიმბოლოიან სუფიქსს. ამასთან უნდა გავითვალისწინოთ ერთი მნიშვნელოვანი გარემოება: სიმბოლოების აღნიშნულ განლაგებას წინ უნდა უძღოდეს იმ სიმბოლოსგან განსხვავებული სიმბოლო, რომელიც წინ უძღოდა k - სიმბოლოიან სუფიქსს.



ტაბულა 10.13

ამ მაგალიში 3-სიმბოლოიან სუფიქსს წინ უძღვის სიმბოლო C, მისი შესაბამისი სიმბოლო ტექსტში არის D და ამ სიმბოლოების შედარებისას მოხდა პირველი არდამთხვევა. ნიმუშში გვაქვს კიდევ ორი ასეთი განლაგება; მარჯვნიდან პირველს წინ უძღვის სიმბოლო C, ხოლო მარჯვნიდან მეორეს - სიმბოლო A. თუ ნიმუშს ისე წავანაცვლებთ, რომ ნიმუშში მარჯვნიდან მეორე BAB განლაგება (რომელსაც წინ ასევე უძღვის სიმბოლო C) დაემთხვეს ტექსტის შესაბამის სიმბოლოებს, აღმოჩნდება, რომ ვიმეორებთ უსარგებლო შედარებას (D ისევ არ დაემთხვევა C-ს).



ტაბულა 10.14

ამ შემთხვევაში, ნიმუშის შესაძლო დასაშვები წანაცვლება იქნება:

ტაბულა 10.15

ანუ, ნიმუშის გადაადგილება ხდება ისე, რომ მარჯვნიდან მეორე BAB განლაგება (რომელსაც წინ არ უძღვის სიმბოლო C) დაემთხვეს ტექსტის შესაბამის სიმბოლოებს.

თუ k - სიმბოლოიანი სუფიქსის ტოლი სიმბოლოების განლაგება ნიმუშში არ არის, მაშინ, თითქოს, ლოგიკურია ვიფიქროთ, რომ წანაცვლება უნდა მოხდეს ნიმუშის ტოლი სიგრძით.

მაგრამ, თუ ნიმუში შეიცავს ელემენტთა თანმიმდევრობას, რომელიც არის k - სიმბოლოიანი სუფიქსის რაიმე l-სიმბოლოიანი სუფიქსი l ¡ k, მაშინ ნიმუშის ტოლი სიგრძით წანაცვლებისას შეიძლება გამოვტოვოთ დასაშვები წანაცვლება.

იმისათვის, რომ ავიცილოთ თავიდან არაკორექტული წანაცვლებები, k სიგრძის სუფიქსისთვის უნდა ვიპოვოთ l¡k სიგრძის უდიდესი პრეფიქსი, რომელიც ემთხვევა, იმავე ნიმუშის l სიგრძის სუფიქსს. თუ ასეთი პრეფიქსი არსებობს, d_2 გამოითვლება როგორც მანძილი პრეფიქსსა და სუფიქსს შორის, წინააღმდეგ შემთხვევაში, d_2 ნიმუშის

სიგრძის ტოლია.

Algorithm 31: Boyer Moore Matcher

Input: ტექსტი T და ტექსტში საძებნი ნიმუში P
Output: ტექსტში ნიმუშის წანაცვლებები

```
1 BOYER-MOORE-HORSPOOL-MATCHER(T, P) :
```

// ბიჯი 1: მოცემული მ სიგრძის ნიმუშისთვის და ანბანისთვის, რომელიც გამოიყენება ტექსტში და ნიმუშში, ავაგოთ წანაცვლებების ცხრილი

// ბიჯი 2: მოცემული მ სიგრძის ნიმუშისთვის, ავაგოთ საერთო სუფიქსების წანაცვლებების ცხრილი

// ბიჯი 3: გავუსწოროთ ნიმუში ტექსტის დასაწყისს

// ბიჯი 4: მანამ, სანამ არ იქნება ნაპოვნი საძიებელი ქვესტრიქონი, ან მანამ, სანამ ნიმუში არ მიაღწევს ტექსტის ბოლო სიმბოლოს, გავიმეოროთ შემდეგი მოქმედებები: ნიმუშის ბოლო სიმბოლოდან დაწყებული, მარჯვნიდან მარცხნივ, ვადარებთ ნიმუშის და ტექსტის შესაბამის ელემენტებს. თუ დადგინდება ყველა m სიმბოლოს ტოლობა (ე.ი. ტექსტში ნაპოვნია ნიმუში) თუ ნიმუშის $0 \leq k < m$ სიმბოლო დაემთხვა ტექსტის შესაბამის სიმბოლოებს, ხოლო $k+1$ სიმბოლო არ დაემთხვა ტექსტის შესაბამის სიმბოლოს და ეს სიმბოლო ტექსტში არის c, მაშინ წანაცვლებების ცხრილიდან ვიპოვით $Table[c]$-ს. თუ $k > 0$, საერთო სუფიქსების წანაცვლებების ცხრილიდან ვიპოვით d_2-ს. ნიმუშს წავანაცვლებთ მარჯვნივ დ პოზიციით, სადაც:

$$d = \begin{cases} d_1 & \text{თუ } k = 0 \\ \max(d_1, d_2) & \text{თუ } k > 0 \end{cases}$$

$$d_1 = \max(Table[c] - k, 1)$$

ალგორითმის მუშაობის დრო. ბოიერ-მურის ალგორითმის მუშაობის დრო უარეს შემთხვევაში არის $\Theta(mn + \Sigma)$. წინასწარი დამუშავების ფაზას სჭირდება $\Theta(mn + \Sigma)$ დრო, ხოლო ძებნის ფაზას - $\Theta(mn)$. ის სწრაფია, როცა ანბანი დიდია (მაგ.: A-Z, 1-9) და ნელია, როცა ანბანი პატარაა (მაგ.: {0,1}).

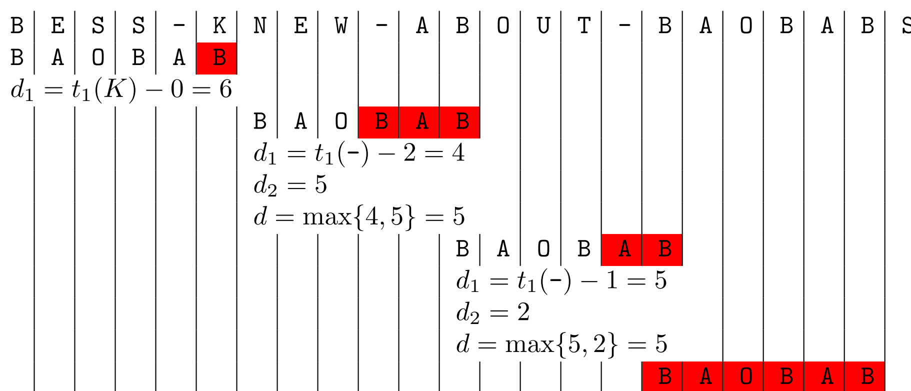
განვიხილოთ ლათინური ანბანის ასოებისგან და ხაზგასმის სიმბოლოებისგან შედგენილ ტექსტში: BESS-KNEW-ABOUT-BAOBABS, ნიმუშის - BAOBAB ძებნის ამოცანა.

c სიმბოლო	A	B	E	S	K	N	W	O	U	T	-
Table[c] წანაცვლება	1	2	6	6	6	6	6	3	6	6	6

(a) წანაცვლებები

k	ნიმუში	d_2
1	$BAO\overline{B}A\underline{B}$	2
2	$\overline{B}AOB\underline{AB}$	5
3	$\overline{B}AO\underline{BAB}$	5
4	$\overline{B}A\underline{OBAB}$	5
5	$\overline{B}\underline{AOBAB}$	5

(b) საერთო სუფიქსების წანაცვლებები



(c) სულ 12 შედარება

ტაბულა 10.16

დასასრულ, განვიხილოთ საკითხი: ანბანის, ტექსტის და ნიმუშის ზომის მიხედვით, ჩვენს მიერ განხილული, რომელი ალგორითმის გამოყენებაა უფრო ხელსაყრელი. არსებობს მრავალი გამოკვლევა ამის შესახებ. ბოიერ-მურის ალგორითმის უპირატესობა უმარტივეს ალგორითმთან შედარებით, განსაკუთრებით საგრძნობია, როცა ანბანიც, ტექსტიც და ნიმუშიც დიდია (გრძელია). ბოიერ-მურის ალგორითმის უპირატესობა მის უფრო მარტივ ბოიერ-მურ-ჰორსპულის ალგორითმთან ვლინდება მხოლოდ მაშინ, როცა ნიმუში გრძელია და ტექსტში ხშირად გვხვდება ნიმუშში შემავალი სიმბოლოების ცალკეული მიმდევრობები. მოკლე ანბანისათვის რეკომენდირებულია კნუტ-მორის-პრატის ალგორითმი, ხოლო თუ ტექსტიც და ნიმუშიც მოკლეა, უმარტივესი ალგორითმიც საკმაოდ ეფექტურია მისი სიმარტივის გამო (წინასწარი დამუშავების ეტაპის გარეშე).

10.7 სავარჯიშოები

1. განვიხილოთ გენების ძებნის ამოცანა დნმ-ს მიმდევრობაში ჰორსპულის ალგორითმის გამოყენებით. დნმ-ს მიმდევრობა წარმოადგენს ტექსტს, განსაზღვრულს {A,C,G,T} ანბანზე, ხოლო გენი, ან გენის მონაკვეთი - ნიმუშს.

   (ა) ააგეთ წანაცვლების ცხრილი გენის შემდეგი მონაკვეთისთვის: TCCTATTC

   (ბ) გამოიყენეთ ჰორსპულის ალგორითმიამ ნიმუშის შემდეგ ტექსტში მოსაძებნად: ATCTGTACTTCCTATTCGTA

2. სიმბოლოების რამდენი შედარება უნდა შესრულდეს ბოიერ-მურ-ჰორსპულის ალგორითმით შემდეგი ნიმუშების ძებნის დროს ტექსტში, რომელიც შედგება 1000 ნულისგან:

   (ა) 00001

(ბ) 10000

(ც) 01010

3. სიმბოლოების რამდენი შედარება უნდა შესრულდეს ბოიერ-მურის ალგორითმით შემდეგი ნიმუშების ძებნის დროს ტექსტში, რომელიც შედგება 1000 ნულისგან:

   (ა) 00001

   (ბ) 10000