ალგორითმები

ლელა ალხაზიშვილი
ალექსანდრე გამყრელიძე

ნახაზები, მაგალითები, დავალებები და LaTeX: ლევან კასრაძე

სარჩევი

თავი 1

მარტივი, რთული, ოპტიმიზაციისა და გადაწყვეტილების ამოცანები

1.1 პოლინომურ დროში ამოხსნადი და მათი „მსგავსი” რთული ამოცანები

- **ეილერის ციკლი გრაფში** (EC)

  მოცემულია: გრაფი $G = (V, E)$.

  პასუხი: „კი” ან „არა”

  განსაზღვრეთ, არსებობს თუ არა მოცემულ გრაფში ისეთი გზა, რომელიც ყველა წიბოზე გაივლის ერთხელ და მხოლოდ ერთხელ?

  ეილერის ციკლის ამოცანა ფართოდ გამოიყენება პრაქტიკაში. მაგალითად, მოცემულია ქალაქის რუკა, რომლის მიხედვითაც დასაგეგმია ნაგვის ან საფოსტო მანქანების მარშრუტი. ცხადია, რომ მანქანამ ქალაქის ყველა ქუჩაზე ერთხელ მაინც უნდა გაიაროს. დროის დასაზოგად უნდა ვიპოვნოთ ისეთი მარშრუტი, რომლითაც ყველა ქუჩაზე ზუსტად ერთხელ გავივლით (თუ ეს შესაძლებელია). შენიშვნა: აქ უნდა ჩავთვალოთ, რომ ყველა ქუჩაზე გავლა მოძრაობის ინტენსიურობაზე დამოკიდებული არაა.

  იგივე ამოცანა შემდეგნაირადაც შეიძლება დავსვათ: შესაძლებელია თუ არა მოცემული გეომეტრიული ფიგურის ხელის აუღებლად ქაღალდზე დახაზვა ისე, რომ უკვე დახაზული აღარ გადავხაზოთ?

  როგორც პირველ სემესტრში ვნახეთ, ამ ამოცანის ამოხსნა რთული არ არის:

  არამიმართულ ბმულ გრაფში არსებობს ეილერის გზა, თუ კენტი ვალენტობის წვეროების რაოდენობა ორზე მეტი არ არის. აღსანიშნავია, რომ თუ გრაფში კენტი ვალენტობის წვერო არ არსებობს, მაშინ იგი ეილერის ციკლს შეიცავს.

  სავარჯიშო 1.1: დაამტკიცეთ, რომ კენტი ვალენტობის მქონე წვეროების რაოდენობა ლუწია.

  თუ გრაფი მიმართულია (და, რა თქმა უნდა, ძლიერად ბმული), მაშინ

  იგი შეიცავს ეილერის ციკლს, თუ მის ყველა წვეროში შემავალი და გამავალი წიბოების რაოდენობა ტოლია;

  იგი შეიცავს ეილერის გზას u წვეროდან v წვეროში, თუ u წვეროს შემავალ წიბოთა რაოდენობა ერთით ნაკლებია გამავალზე, ხოლო v წვეროსთვის კი პირიქით: შემავალ წიბოთა რაოდენობა ერთით მეტია გამავალზე, ხოლო ყველა დანარჩენ წვეროში შემავალ და გამომავალ წიბოთა რაოდენობა ტოლია.

  სავარჯიშო 1.2: დაამტკიცეთ ზემოთ მოყვანილი გამონათქვამები (იხ. პირველი სემესტრის მასალა).

  სავარჯიშო 1.3: დაწერეთ ალგორითმი, რომელიც მოცემული გრაფისთვის ეილერის ციკლის არსებობას დაადგენს.

  თუ გრაფში ეილერის ციკლი (ან გზა) არსებობს, მისი დადგენაც სწრაფად შეიძლება: პირველ რიგში უნდა ვიპოვნოთ *ნებისმიერი* ციკლი, დავიხსომოთ და საწყისი გრაფიდან მისი წიბოები ამოვშალოთ. ცხადია, რომ ამოშლის შემდეგ დარჩენილ გრაფში ეილერის ციკლის არსებობის პირობა შენარჩუნებული იქნება (რადგან ციკლში შემავალი წვეროების ვალენტობა ორით შემცირდება). იგივე პროცედურა რეკურსიულად

გავიმეოროთ მანამ, სანამ გრაფში წიბოები არ გამოილევა. დაგვრჩება საწყის გრაფში არსებული ციკლების სიმრავლე, რომელთა გაერთიანებაც მთელს გრაფს მოგვცემს.

სავარჯიშო 1.4: დაამტკიცეთ, რომ მიღებული სიმრავლის ციკლების გაერთიანება მართლაც მთელ საწყის გრაფს მოგვცემს და ნებისმიერი ორი ციკლი საერთო წიბოს არ შეიცავს.

სავარჯიშო 1.5: დაწერეთ ალგორითმი, რომელიც ზემოთ აგებული ციკლების სიმრავლისგან ეილერის გზას გამოითვლის და შეაფასეთ მისი ბიჯების რაოდენობის ზედა ზღვარი.

ეილერის ციკლის ალგორითმი ბევრ პრაქტიკულ (მათ შორის ზემოთ ხსენებულ ნაგვის შეგროვების) ამოცანას გადაგვიჭრიდა, მაგრამ, სამწუხაროდ, პრაქტიკაში არსებული ამოცანების შესაბამის გრაფებში უმეტეს შემთხვევაში ეილერის ციკლის პირობა არ კმაყოფილდება. ასეთ შემთხვევაში დასმულია უფრო ზოგადი

ჩინელი ფოსტალიონის ამოცანა მოცემულ G გრაფში იპოვნეთ ისეთი უმოკლესი მარშრუტი, რომელიც ყველა წიბოზე ერთხელ მაინც გადაივლის.

ამ ამოცანის გადასაჭრელად პირველ რიგში უნდა შევქმნათ სრული შეწონილი $G' = (V', E')$ გრაფი, რომელიც G გრაფის კენტიანი წვეროებისგან შედგება და $(u, v) \in E'$ წიბოს წონა G გრაფის შესაბამის წვეროებს შორის მინიმალური გზის სიგრძის ტოლია. G' გრაფზე ოპტიმალური დაწყვილების ამოცანის (perfect matching) ამოხსნა (ანუ ისეთი წიბოების სიმრავლის გამოყოფა, რომლებითაც მთელი გრაფი გადაიფარება და არც ერთ წყვილს საერთო წვერო არ აქვს, თან ასეთ სიმრავლეებს შორის ვირჩევთ ისეთს, რომელთა წიბოების წონების ჯამი ინიმალურია) გვაძლევს იმ წიბოთა სიმრავლეს, რომელიც უნდა დაემატოს G გრაფს. ამ სახით შექმნილ G_1 გრაფზე ეილერის ციკლი უნდა არსებობდეს, რადგან ყოველ ორ კენტიან წვეროს ვაერთებთ წიბოთი და ყველა წვერო ლუწიანი გამოვა.

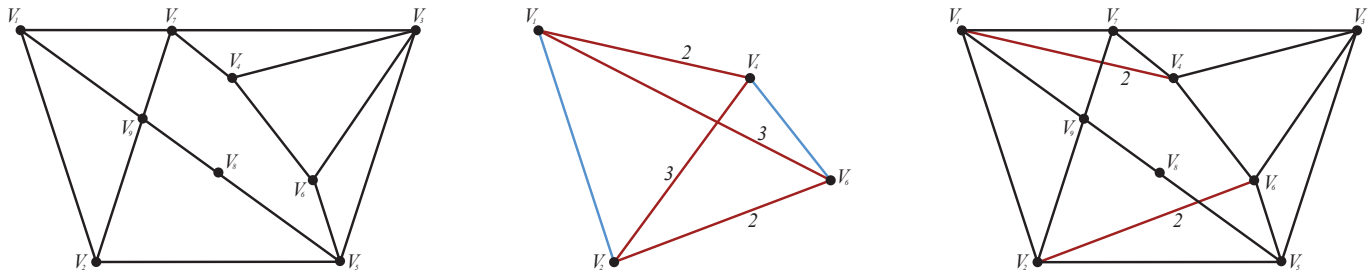
სავარჯიშო 1.6: დაწერეთ მოცემული გრაფის ოპტიმალური დაწყვილების გამოთვლის ალგორითმი და შეაფასეთ მისი ბიჯების რაოდენობის ზედა ზღვარი.

შენიშვნა: აქ გასათვალისწინებელია ის ფაქტი, რომ G' გრაფის ოპტიმალურ დაწყვილებაში G გრაფში არსებული წიბოები არ უნდა შედიოდეს.

თუ G_1 გრაფზე ეილერის ციკლში „ახალ" წიბოებს შესაბამისი უმოკლესი გზით შევცვლით, მივიღეთ უმოკლეს გზას, რომელიც ყველა წიბოზე ერთხელ მაინც გადაივლის.

რაც შეეხება მიმართულ გრაფს, G' გრაფის წიბოები უნდა იყოს მიმართული ნაკლებ გამომავალი წიბოების მქონე წვეროდან ნაკლებ შემავალი წიბოების მქონე წვეროსკენ, რითაც შეიქმნება ორად დაყოფილი მიმართული გრაფი და ოპტიმალური დაწყვილების ამოცანაც მასზე უნდა გადაიჭრას.

მაგალითისთვის განვიხილოთ **??** ნახაზში მოყვანილი G (ა), G' (ბ) და G_1 (გ) გრაფები.



ჩინელი ფოსტალიონის ამოცანის ამოსახსნელი გრაფები

სავარჯიშო 1.7: **??** ნახაზში მოყვანილი G_1 გრაფის მეშვეობით გამოიანგარიშეთ G გრაფის ოპტიმალური მარშრუტი.

სავარჯიშო 1.8: დაამტკიცეთ, რომ ზემოთ მოყვანილი მეთოდით გამოთვლილი გზა მინიმალური იქნება.

სავარჯიშო 1.9: დაწერეთ ალგორითმი, რომელიც ჩინელი ფოსტალიონის ამოცანას გადაჭრის.

სავარჯიშო 1.10: რა სიგრძის იქნება ჩინელი ფოსტალიონის ამოცანის ამოხსნა ხეებისთვის?

ეილერის ციკლის პოვნის პარადიგმა ძალიან მნიშვნელოვანია გრაფებზე პარალელური ალგორითმების აგებაში. ხშირად გრაფზე აგებენ დამფარავ ხეებს და მერე იყენებენ ეილერის ციკლების ტექნიკას მათი დამუშავებისთვის.

ამას გარდა, მნიშვნელოვანია გრაფში ეილერის ციკლების რაოდენობის გამოთვლაც.

- **ჰამილტონის ციკლი გრაფში** (HC)

  მოცემულია: გრაფი $G = (V, E)$.

  პასუხი: „კი” ან „არა”

  განსაზღვრეთ, არსებობს თუ არა მოცემულ გრაფში ისეთი გზა, რომელიც ყველა წვეროზე გაივლის ერთხელ და მხოლოდ ერთხელ?

  ერთი შეხედვით ეს ამოცანა ეილერის ციკლის ამოცანის მსგავსია და შეიძლება ვიფიქროთ კიდეც, რომ უფრო მარტივად ამოიხსნაა შესაძლებელი. მაგრამ მისთვის პოლინომურ დროში მომუშავე ალგორითმი ჯერ-ჯერობით არაა ცნობილი. უფრო მეტიც: როგორც შემდგომში ვნახავთ, არც ისაა ცნობილი, შესაძლებელია თუ არა ამ ამოცანისთვის ამდაგვარი ალგორითმის დაწერა.

  ანალოგიურად შეიძლება ჩამოვაყალიბოთ ჰამილტონის გზის ამოცანაც (იგივე ჰამილტონის ციკლი, მხოლოდ საწყის წვეროში დაბრუნება აღარაა საჭირო).

  ჰამილტონის გზის ამოცანა უაღრესად მნიშვნელოვანია პრაქტიკაშიც. ასე, მაგალითად, გენურ ინჟინერიაში აქვთ გენეტური კოდის ნაწილები $a_1, ..., a_n$ და წესები, რომელი ნაწილი რომელს შეიძლება გადაებას. თუ ამ კოდებს განვიხილავთ, როგორც გრაფის წვეროებს და ერთმანეთზე გადაბმად ნაწილებს შორის გავავლებთ (მიმართულ) წიბოს, ჰამილტონის ციკლი გვიჩვენებს იმ გენეტურ კოდს, რომელიც შეიძლება შეიქმნას მოცემული ნაწილებიდან.

  მეორე გამოყენება შეიძლება იყოს სკოლის ავტობუსის მარშრუტის დაგეგმვა: თუ მოცემული გვაქვს მოსწავლეების საცხოვრებელი მისამართები (გრაფის წვეროები) და შემაერთებელი გზები (გრაფის წიბოები), ჰამილტონის (მინიმალური) ციკლი ოპტიმალურ მარშრუტს მოგვცემს.

  გარდა ამისა, როგორც ვნახანთ, ჰამილტონის ციკლის ამოცანა ე.წ. NP-სრულ ამოცანათა კლასს განეკუთვნება, რაც იმას ნიშნავს, რომ მისი სწრაფად ამოხსნა ჩვენს გარშემო არსებული ამოცანების უმეტესობის სწრაფად ამოხსნის გზას გვაჩვენებდა.

- **უმოკლესი გზა გრაფის წვეროებს შორის**

  მოცემულია: შეწონილი გრაფი $G = (V, E)$ და მისი ორი წვერო $u, v \in V$.

  პასუხი: მიმდევრობა $(x_1, ..., x_k)$, სადაც $x_i \in V$, $x_i \neq x_j$, თუ $i \neq j$.

  იპოვნეთ უმოკლესი გზა გრაფში მოცემულ ორ წვეროს შორის.

  უმოკლესი გზის ამოცანა უაღრესად მნიშვნელოვანია პრაქტიკაში, მაგ. მარშრუტიზაციის ამოცანებში.

  ამ ამოცანისთვის სწრაფი ალგორითმი იხ. მეორე სემესტრის კონსპექტში.

- **მაქსიმალური მარტივი გზა გრაფის წვეროებს შორის**

  მოცემულია: შეწონილი გრაფი $G = (V, E)$ და მისი ორი წვერო $u, v \in V$.

  პასუხი: მიმდევრობა $(x_1, ..., x_k)$, სადაც $x_i \in V$, $x_i \neq x_j$, თუ $i \neq j$.

  იპოვნეთ მაქსიმალური მარტივი გზა ამ ორ წვეროს შორის (მარტივი ეწოდება ისეთ გზას, რომელიც ციკლებს არ შეიცავს).

  არც ამ ამოცანისთვისაა პოლინომურ დროში მომუშავე ალგორითმი ცნობილი და არც არავინ იცის, შესაძლებელია თუ არა ამ ამოცანისთვის ამდაგვარი ალგორითმის დაწერა.

  ერთი შეხედვით ეს ამოცანა პრაქტიკულ გამოყენებას მოკლებულია, მაგრამ თუ დავუშვებთ, რომ ერთი წვეროდან მეორეზე გადასვლა რაიმე ეკონომიკურ ტრანსაქციასთანაა დაკავშირებული და გადასული წიბოს წონა ფინანსურ მოგებას (ან უარყოფითი წონის შემთხვევაში წაგებას) აღნიშნავს, მაშინ მაქსიმალური გზა მაქსიმალური მოგების ტოლფასია.

  აქვე უნდა აღინიშნოს ამ ამოცანის ჰამილტონის ციკლთან კავშირი: თუ არაშეწონილ გრაფში ჰამილტონის ციკლი არსებობს, მაშინ სწორედ ეს იქნება მაქსიმალური გზა.

  სავარჯიშო 1.11: როგორ შეიძლება მაქსიმალური გზის ამოცანის ამოხსნით ჰამილტონის ციკლის (და, შესაბამისად, გზის) ამოცანის გადაჭრა?

- **უდიდესი სრული ქვეგრაფი** *Clique*

  მოცემულია: გრაფი $G = (V, E)$.

  პასუხი: $k \in \mathbb{N}$.

  რა არის ისეთ წვეროთა მაქსიმალური ქვესიმრავლის ზომა, რომელშიც ნებისმიერი ორი წვერო ერთმანეთთან იქნება დაკავშირებული (ანუ G გრაფის ყველაზე დიდი სრული ქვეგრაფის წვეროების რაოდენობა)

  ეს ამოცანაც რთულია: მისთვის სწრაფი ალგორითმი ცნობილი არაა (და არც ისაა ცნობილი, ამოხსნადია თუ არა სწრაფად)

  უდიდესი საერთო ქვეგრაფის ამოცანა (შემოკლებით კლიკი, Clique) მსგავსი ობიექტების (კლასტერების) აღმოჩენაში უმთავრეს როლს თამაშობს. მაგალითად, თუ საჭიროა გაყალბების აღმოჩენა დოკუმენტების (ფულის, ფასიანი ქაღალდების, სადაზვეო მაქინაციების და სხვა) დიდ სიმრავლეში, პირველ რიგში უნდა განისაზღვროს „მსგავსება" (როგორც წესი, ეს ე.წ. „ჰემინგის მანძილის" ან შესაბამის ვექტორებს შორის კუთხის კოსინუსით ხდება). არსებულ დოკუმენტებს თუ განვიხილავთ, როგორც გრაფის წვეროებს, ხოლო მსგავს დოკუმენტებს წიბოებით შევაერთებთ (რაც უფრო დიდია მსგავსების კოეფიციენტი, მით უფრო მოკლეა მათ შორის გზა). როგორც წესი, არსებულ დოკუმენტებს (მაგ. ფულის ნომრებს) შორის დიდი მსგავსება არ უნდა იყოს, მაგრამ დიდი რაოდენობის დოკუმენტების გაყალბების შემთხვევაში მსგავსება უფრო დიდია ხოლმე. აქედან გამომდინარე, უდიდესი სრული ქვეგრაფი „მსგავს" ობიექტების დიდ რაოდენობას იპოვნის, რისი უფრო დაწვრილებითი ანალიზის შემდეგ გაყალბების პოვნა უფრო მარტივი იქნება.

  სწორედ ამ იდეებზე დაყრდნობით განავითარა ამერიკის გადასახადების სამსახურმა ყალბი დოკუმენტაციის აღმოჩენის სისტემა.

  როგორც ვთქვით, ამ ამოცანისთვის პოლინომური ალგორითმი არ არის ცნობილი (და არც ის ვიცით, შესაძლებელია თუ არა ასეთის არსებობა). როგორც შემდგომში ვნახავთ, ეს ამოცანაც ე.წ. NP-სრულ ამოცანათა კლასს განეკუთვნება და, როგორც გამოცდილებამ აჩვენა, მისი ზუსტად ამოხსნის ნაცვლად სხვა გზებია მოსაძებნი.

  ერთ-ერთი პირველი გამოსავალია არა *უდიდესი*, არამედ (ლოკალურად) მაქსიმალური სრული ქვეგრაფის მოძებნა, რაც ხშირად პრაქტიკულ ამოცანებში საკმარისია ხოლმე. ლოკალურად მაქსიმალური ეწოდება ისეთ სრულ ქვეგრაფს, რომლისთვისაც საწყისი გრაფის ნებისმიერი სხვა წვეროს მიერთება სრულობას დაუკარგავს.

  ლოკალურად მაქსიმალური სრული ქვეგრაფის საპოვნელად შეიძლება წვეროების დალაგება მათი ვალენტობის კლებადობის მიხედვით, პირველ (მაქსიმალური ვალენტობის მქონე) წვეროს კლიკის სიმრავლეში ჩავრთავთ, შემდეგ რიგ-რიგობით ჩავყვებით სიას და ჩავრთავთ იმ წვეროებს, რომლებიც სრულ გრაფს შეგვიქმნის. თუ შესაბამისი წვეროს კლიკთან მიკუთვნებას ერთი ბიტით აღვნიშნავთ და მათ ყველას ე.წ. ბიტ-ვექტორში გავაერთიანებთ, ამ ალგორითმის რეალიზაცია $O(|V| + |E|)$ დროში იქნება შესაძლებელი.

  სავარჯიშო 1.12: დაწერეთ პროგრამა, რომელიც ლოკალურად მაქსიმალურ სრულ ქვეგრაფს წრფივ დროში გამოითვლის.

  მეორე ხერხი, რაც შეიძლება პრაქტიკაში გამოგვადგეს, არა *სრული,* არამედ *მჭიდრო* ქვეგრაფების ძებნაა. ხშირად პრაქტიკულ ამოცანებში სრული ქვეგრაფების ნაცვლად მჭიდრო ქვეგრაფის მოძებნაც უკვე საკმარისია. აღსანიშნავია, რომ სრული ქვეგრაფი ამავდროულად მაქსიმალურად მჭიდრო ქვეგრაფია.

  მინიმუმ k რიგის (ვალენტობის) ქვეგრაფის პოვნა (ანუ ისეთის, რომლის წვეროების რიგიც არის მინიმუმ k) ადვილად შეიძლება: გრავში ამოვაგდებთ იმ წვეროებს (და შესაბამის წიბოებს), რომელთა რიგიც უფრო დაბალია. ამ პროცედურამ შეიძლება გამოიწვიოს დარჩენილი წვეროების რიგის შემცირება (თუ ისინი საკმარისად ბევრ ამოგდებულ წვეროსთან იყვნენ მიერთებულნი). ამ პროცედურას ვიმეორებთ მანამ, სანამ არ დაგვრჩება მხოლოდ შესაბამისად მაღალი რიგის წვეროები.

  სავარჯიშო 1.13: დაწერეთ ალგორითმი $O(|V| + |E|)$ რიგის ალგორითმი, რომელიც k-მჭიდრო ქვეგრაფს გამოითვლის.

  შენიშვნა: გამოიყენეთ გრაფის მეზობლობის სიით აღწერა და შეზღუდული ზომის პრიორიტეტული რიგი.

  ბოლოს უნდა აღინიშნოს, რომ თუ გრაფი პლანარულია, მასში მაქსიმალური სრული ქვეგრაფი 2, 3 ან 4 წვეროსგან უნდა შედგებოდეს (ტოპოლოგიის ერთ-ერთი თეორემა გვეუბნება, რომ K_5, ანუ 5 წვეროიანი სრული გრაფი პლანარული ვერ იქნება).

  სავარჯიშო 1.14: დაწერეთ ალგორითმი, რომელიც $O(|V|)$ დროში გამოითვლის პლანარული გრაფის მაქსიმალურ ქვეგრაფს.

  რაც შეეხება *Clique* ამოცანის ზოგად ამოხსნას, მისთვის მრავალი სხვადასხვა მიდგომაა განვითარებული, რაც ძირითადად ალბათური ევრისტიკის იდეებს ეყრდნობა, მაგრამ ეს თემა ჩვენი ამ სემესტრის კურსის ფარგლებს ცდება და შემდეგში უნდა იქნას განხილული.

- **იზოლირებული წვეროების სიმრავლე** IS

  მოცემულია: გრაფი $G = (V, E)$.

  პასუხი: $k \in \mathbb{N}$.

  რა არის ისეთ წვეროთა მაქსიმალური ქვესიმრავლის ზომა, რომელშიც ნებისმიერი ორი წვერო ერთმანეთთან არ იქნება დაკავშირებული (ანუ G გრაფის ყველაზე დიდი იზოლირებული წვეროების სიმრავლის ელემენტების რაოდენობა)

- **გრაფის გადაფარვა** VC

  მოცემულია: გრაფი $G = (V, E)$.

  პასუხი: $k \in \mathbb{N}$.

  რა არის ისეთ წვეროთა მინიმალური ქვესიმრავლის ზომა, რომლითაც გრაფის ყველა წიბო გადაიფარება? (თუ e წიბო v წვეროს შეიცავს, მაშინ ამბობენ, რომ v გადაფარავს e წიბოს)

- **გრაფის ქრომატული რიცხვი** CN

  მოცემულია: გრაფი $G = (V, E)$.

  პასუხი: $k \in \mathbb{N}$.

  მოცემული გრაფისთვის განსაზღვრეთ, მინიმუმ რამდენ ფრად შეიძლება მისი შეღებვა.

  განმარტება 1.1: მოცემულია გრაფი $G = (V, E)$. მისი **შეღებვა** ეწოდება ისეთ $\phi : V \to \mathbb{N}$ ასახვას, რომელიც ყოველ წვეროს რაიმე ნატურალურ რიცხვს (ფერს) შეუსაბამებს ისე, რომ მეზობლებს (წიბოთი შეერთებულ წვეროებს) ერთი და იგივე ფერი (ერთი და იგივე რიცხვი) არ შეესაბამებოდეს: $(u, v) \in E \Rightarrow \phi(u) \neq \phi(v)$.

ძალიან მნიშვნელოვანია შემდეგი ორი ამოცანა:

განმარტება 1.2: $G_1 = (V_1, E_1)$ და $G_2 = (V_2, E_2)$ გრაფს ეწოდება ერთმანეთის იზომორფული (აღინიშნება $G_1 \cong G_2$), თუ მოიძებნება ისეთი ბიექცია $f : V_1 \to V_2$, რომ $(v, u) \in E_1 \Leftrightarrow (f(v), f(u)) \in E_2$ (ანუ G_1 გრაფში ორი წვერო შეერთებულია წიბოთი მაშინ და მხოლოდ მაშინ, თუ მეორე გრაფში მათი სახეებია შეერთებული წიბოთი).

განმარტება 1.3: $G' = (V', E')$ გრაფს ეწოდება $G = (V, E)$ გრაფის ქვეგრაფი, თუ $V' \subset V$ და $(u, v) \in E' \Rightarrow (u, v) \in E$.

- **გრაფთა იზომორფიზმი** GI

  მოცემულია: ორი გრაფი G_1, G_2

  პასუხი: „კი” ან „არა”

  განსაზღვრეთ, არსებობს თუ არა მოცემულ გრაფებს შორის ერთმანეთის იზომორფიზმი, ანუ $G_1 \cong G_2$

- **ქვეგრაფის იზომორფიზმი** SGI

  მოცემულია: ორი გრაფი G_1, G_2

  პასუხი: „კი” ან „არა”

  განსაზღვრეთ, მოიძებნება თუ არა G_1 გრაფის ისეთი G' ქვეგრაფი, რომ $G' \cong G_2$?

- **კომი ვოიაჟერის (მოგზაური ვაჭრის) ამოცანა** TSP

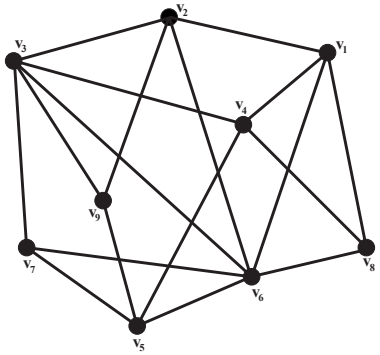
  მოცემულია: შეწონილი გრაფი G

  პასუხი: $k \in \mathbb{R}$

  განსაზღვრეთ ისეთი მინიმალური მარშრუტი, რომელიც ყველა წვეროზე ერთხელ მაინც გაივლის

სავარჯიშო 1.15: განვიხილოთ გრაფი G, რომელიც მოცემულია შემდეგ ნახაზში.
სავარჯიშო გრაფი
მასში შეიძლება გამოიყოს სამ ელემენტიანი იზოლირებული სიმრავლე $\{v_7, v_9, v_4\}$. შესაძლებელია თუ არა მასში უფრო დიდი იზოლირებული სიმრავლის პოვნა? რას მოგვცემს პასუხად $IS(G)$?

სავარჯიშო 1.16: ზემოთ მოყვანილი G გრაფისთვის რა იქნება $Clique(G)$? $CN(G)$? $HC(G)$? $VC(G)$? $TSP(G)$?



არსებობს თუ არა მასში ეილერის ციკლი? (პასუხი დაამტკიცეთ)

სავარჯიშო 1.17: იგივე G გრაფში იპოვნეთ მაქსიმალური მარტივი გზა v_1 და v_2 წვეროებს შორის.

სავარჯიშო 1.18: C++ ენაზე დაწერეთ პროგრამა, რომელიც მატრიცის სახით მოცემული გრაფისთვის გადაჭრის ეილერის ციკლის ამოცანას.

პარაგრაფის ბოლოს კვლავ მოვიყვანოთ რამოდენიმე მნიშვნელოვანი ამოცანა.

- **ნატურალური რიცხვის სიმარტივის ტესტი**

  მოცემულია: ნატურალური რიცხვი $n \in \mathbb{N}$.

  პასუხი: „კი" ან „არა"

  დაადგინეთ, მარტივია თუ არა n.

  შენიშვნა: ეს ამოცანა ანტიკური დროიდან მოყოლებული რთულად ითვლებოდა. მისი უდიდესი პრაქტიკული გამოყენების (მაგალითად, კრიპტოგრაფიაში) მიუხედავად, სწრაფი ალგორითმი ცნობილი არ იყო, სანამ 2002 წელს ინდოელმა ინფორმატიკოსებმა $O(k^{12})$ ზედა ზღვრის მქონე ალგორითმი შეიმუშავეს, რომელიც შემდგომში კიდევ უფრო გაუმჯობესდა. დაწვრილებითი ინფორმაციისთვის იხილეთ

  Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, PRIMES is in P

  Annals of Mathematics, vol. 160, pp. 781 - 793, Springer Verlag, 2004

  აღსანიშნავია, რომ ალგორითმის დროის ზედა ზღვარი ყოველთვის მონაცემის სიგრძეზეა დამოკიდებული, ამიტომაც ზემოთ მოყვანილ ფორმულაში ვიხმარეთ პარამეტრი k, რომელიც მოცემული n რიცხვის ბიტების რაოდენობას აღნიშნავს: $k = \log n$.

- **გრაფის პლანარულობა**

  მოცემულია: გრაფი G

  პასუხი: „კი" ან „არა"

  დაადგინეთ, არის თუ არა იგი პლანარული?

ისეთი ამოცანების მაგალითებად, რომელთათვისაც რეალურ დროში ამოხსნადი ალგორითმი ცნობილი არაა (მაგრამ არც ისაა დამტკიცებული, რომ მათთვის ასეთი ალგორითმი არ არსებობს), შეიძლება მოვიყვანოთ:

- **რიცხვის ფაქტორიზაცია**

  მოცემულია: ნატურალური რიცხვი $n \in \mathbb{N}$

  პასუხი: ნატურალურ რიცხვთა მიმდევრობა $p_1, ..., p_h$

  დაშალეთ n მარტივ მამრავლებად: $n = p_1, ..., p_h$

- **გრაფის ოპტიმალურად დახატვა**

  მოცემულია: გრაფი G

  პასუხი: გრაფის ნახაზი სიბრტყეზე

  დახაზეთ გრაფი სიბრტყეზე ისე, რომ წიბოების გადაკვეთის რაოდენობა მინიმალური იყოს.

- **ბულის ფუნქციათა ჭეშმარიტება (SAT)**

  მოცემულია: ბულის ალგებრის n ცვლადზე დამოკიდებული ფუნქცია $f : \mathbb{B}^n \to \mathbb{B}$, რომელიც ჩაწერილია კონიუნქციური ნორმალური ფორმით

  პასუხი: „კი" ან „არა"

  არსებობს თუ არა მისი ცვლადების ისეთი მნიშვნელობები, რომ ეს ფუნქცია იყოს ჭეშმარიტი (ანუ იღებდეს მნიშვნელობას 1)?

1.2 ოპტიმიზაციისა და გადაწყვეტილების ამოცანები

ადვილი შესამჩნევია, რომ აქ მოყვანილ ამოცანებში ზოგიერთის პასუხია „კი" ან „არა" და ზოგიერთის – რაღაც კონკრეტული რიცხვი ან სხვა მათემატიკური სტრუქტურა, რომელიც გარკვეულ წინაპირობას აკმაყოფილებს.
ისეთ ამოცანებს, რომლთა პასუხი შეიძლება იყოს „კი„ ან „არა", ***გადაწყვეტილების ამოცანა*** ეწოდება, ხოლო მეორე ტიპისას კი - ***ოპტიმიზაციის***.
გადაწყვეტილების ამოცანებია, მაგალითად, ჰამილტონის ციკლის, ეილერის ციკლის, გრაფის პლანარულობის ტესტის, სიმარტივის ტესტის, გრაფის იზომორფიზმის, ქვეგრაფის იზომორფიზმის და სხვა ამოცანები. მათში იმის დადგენაა საჭირო, ეკუთვნის თუ არა მონაცემი ამა თუ იმ კლასს: თუ მოცემული გვაქვს ყველა შესაძლო პლანარული გრაფის სიმრავლე, რომელსაც ჩვენ პირობითად ვუწოდებთ G_P, მოცემული G გრაფისთვის მხოლოდ შემდეგი საკითხის გარკვევაა საჭირო: $G \in G_P$?
ასევე, ჰამილტონის ციკლის მქონე გრაფთა სიმრავლე თუ აღინიშნება როგორც G_H, ჰამილტონის ციკლის ამოცანა იგივეა, რაც შეკითხვა $G \in G_H$? ამიტომ ამდაგვარ ამოცანებს უწოდებენ „გადაწყვეტილების ამოცანას": გადაწყვიტეთ, ეკუთვნის თუ არა მონაცემი რაღაც სიმრავლეს (ატარებს თუ არა გარკვეულ თვისებას).

ამისგან განსხვავებულია, მაგალითად, გადაფარვის ამოცანა VC. აქ ყველა შესაძლო გადაფარვიდან უნდა ავირჩიოთ მინიმალური. ასევე CN შეღებვის ამოცანაც: ყველა შესაძლო შეღებვიდან მინიმალური; TSP: ყველა შესაძლო მარშრუტიდან, რომელიც ყველა ქალაქზე გადის, მინიმალური და ა.შ. სხვა სიტყვებით რომ ვთქვათ, აქ შესაძლო ამონახსნებიდან ვირჩევთ რაღაც თვალსაზრისით ოპტიმალურს. ამიტომაც ასეთი ტიპის ამოცანებს უწოდებენ „ოპტიმიზაციისას".

განმარტება 1.4: თუ მოცემულია რაიმე გადაწყვეტილების ამოცანა A მონაცემთა სიმრავლით X, მაშინ $L_A = \{x \mid x \in X,\ A(x) =$კი$\}$ მონაცემთა ქვესიმრავლეს (ისეთ მონაცემთა სიმრავლეს, რომლებზეც ამ ამოცანის პასუხი იქნებოდა „კი") ამ A ამოცანის ***ენა*** ეწოდება.

სავარჯიშო 1.19: სიტყვიერად ახსენით, რა არის $L_{EC}, L_{TSP_k}, L_{VC_k}$ და L_{CN_k}.

საინტერესოა კავშირი ოპტიმიზაციისა და გადაწყვეტილების ამოცანებს შორის.
წარმოვიდგინოთ, რომ გვაქვს შემდეგი გადაწყვეტილების ამოცანა:

გრაფის k ფრად შეღებვა (CN_k)
მოცემულია: გრაფი G და რიცხვი $k \in \mathbb{N}$
პასუხი: „კი" ან „არა"
გაეცით პასუხი შეკითხვას: შეიძლება თუ არა G გრაფის k ფრად შეღებვა?

თუ გვაქვს ამ ამოცანის ალგორითმი, ადვილად შევძლებთ CN ოპტიმიზაციის ამოცანის გადაჭრასაც:

ალგორითმი 1: გრაფის შეღებვის ალგორითმი
მოცემულია: გრაფი G

```
1: k = 1;
2: while(CN_j(G) = „არა");
3: k++
4: return(k)
```

ანალოგიურად, თუ გვაქვს VC_k გადაწყვეტილების ამოცანის ალგორითმი, რომელიც მოგვცემს პასუხს „კი" მაშინ და მხოლოდ მაშინ, თუ მოცემული გრაფი გადაიფარება k წვეროთი, მისი მეშვეობით ადვილად შეიძლება გადაიჭრას შესაბამისი ოპტიმიზაციის ამოცანაც.

სავარჯიშო 1.20: ჩამოაყალიბეთ IS, $Clique$, TSP და VC შესაბამისი გადაწყვეტილების ამოცანები.

სავარჯიშო 1.21: მოცემულია IS_k, $Clique_k$, TSP_k და VC_k გადაწყვეტილების ამოცანების ალგორითმები. მათი გამოყენებით დაწერეთ შესაბამისი ოპტიმიზაციის ამოცანების ალგორითმები.

1.3 არაამოხსნადი ამოცანები

ერთ-ერთი ყველაზე ძველი ამოცანა, რომელსაც ალგორითმული ამოხსნა არ აქვს (გარკვეული რესურსებით შეზღუდვის გათვალისწინებით), ანტიკური საბერძნეთიდან მოდის და რამოდენიმე ათასი წელი გადაუჭრელი რჩებოდა - ფიგურების ფარგლითა და სახაზავით აგება (იხ. პირველი სემესტრის კურსი „ალგორითმების შესავალი"). როგორც XIX საუკუნეში მათემატიკოსებმა დაამტკიცეს, შეუძლებელია ისეთი ალგორითმების პოვნა, რომლითაც ერთეულოვანი წრის ფართობის მქონე კვადრატის აგებას შევძლებდით ფარგლისა და სახაზავის გამოყენებით. სამაგიეროდ შეიძლება ასეთი ფართობის მქონე კვადრატს ნებისმიერი სიზუსტით მივუახლოვდეთ - ამოცანის ოდნავი შეცვლისას იგი ამოხსნადი ხდება: მოცემულია წრე ფართობით S_1 და რაიმე დადებითი რიცხვი $\epsilon > 0$ (სიზუსტე). ფარგლისა და სახაზავის გამოყენებით ააგეთ ისეთი კვადრატი, რომლის ფართობი S_2 მოცემული სიზუსტით იქნება მოცემული წრის ფართობის სიახლოვეში, ანუ $|S_1 - S_2| \leq \epsilon$.
მეორე მაგალითად შეიძლება ჰილბერტის X პრობლემის მოყვანა: მოცემულია ნებისმიერი დიოფანტეს პოლინომიური განტოლება მთელი კოეფიციენტებით: $a_n \cdot x_n^n + \cdots + a_1 \cdot x_1^1 + a_0 = 0$, სადაც $a_i \in \mathbb{Q}, 0 \leq i \leq n, n \in \mathbb{N}$. შეიმუშავეთ მეთოდი (ანუ ალგორითმი), რომელიც ნებისმიერი ასეთი პოლინომისათვის გაგვცემდა პასუხს (კი ან არა) შეკითხვაზე, აქვს თუ არა ამ განტოლებას მხოლოდ რაციონალური ფესვები? როგორც აღმოჩნდა, არც ამ ამოცანას აქვს ალგორითმული ამოხსნა (აქვე შევნიშნოთ, რომ გარკვეულ კონკრეტულ შემთხვევებში ეს საკითხი გადაიჭრება. აქ ლაპარაკია იმაზე, რომ არ არსებობს ისეთი ალგორითმი, რომელიც ნებისმიერი მონაცემისთვის გაგვცემდა პასუხს).
აღსანიშნავია, რომ ეს ამოცანები ალგორითმის განსაზღვრებაზე ადრე ჩამოყალიბდა და წმინდა მათემატიკური ხერხებით იქნა გადაჭრილი, თუმცა მათ ინფორმატიკაში ძალიან მნიშვნელოვანი როლი ითამაშეს.
პირველი ამოცანა (ანუ იგივე პრობლემა), რომლის ამოუხსნადობაც ალგორითმულ მეთოდებზე დაყრდნობით დამტკიცდა, ე.წ. შეჩერების ამოცანაა: ნებისმიერი ალგორითმისათვის (პროგრამისთვის) და მისი მონაცემისათვის დაადგინეთ, შეჩერდება თუ ჩაიციკლება ეს ალგორითმი მოცემულ მონაცემზე. ამის კონკრეტული მაგალითია სამი პროგრამა, რომელთაგან პირველი რიგ-რიგობით გადაამოწმებს ნატურალურ რიცხვებს და შეჩერდება მაშინ და მხოლოდ მაშინ, თუკი ისეთ ლუწ რიცხვს აღმოაჩენს, რომელიც ორი სხვა ლუწი რიცხვის ჯამს წარმოადგენს.

ალგორითმი 2: ლუწ რიცხვთა შემოწმება (პირველი ვერსია)

```
1: n = 0;
2: do
3: n++;
4: while(2n ≠ 2k + 2p)
```

მეორე ალგორითმი რიგ-რიგობით გადაამოწმებს ნატურალურ რიცხვებს და შეჩერდება მაშინ და მხოლოდ მაშინ, თუ ისეთ კენტ რიცხვს აღმოაჩენს, რომელიც ერთი ლუწი და ერთი კენტი რიცხვის ჯამს წარმოადგენს.

ალგორითმი 3: ლუწ რიცხვთა შემოწმება (მეორე ვერსია)

```
1: n = 0;
2: do
3: n++;
4: while(2n ≠ 2k + (2p − 1))
```

ცხადია, რომ პირველ ალგორითმზეც და მეორეზეც წინასწარ შეიძლება იმის თქმა, შეჩერდება თუ არა იგი.
მაგრამ განვიხილოთ ალგორითმი, რომელიც რიგ-რიგობით გადაამოწმებს ლუწ რიცხვებს და შეჩერდება მაშინ და მხოლოდ მაშინ, თუ განხილული რიცხვი მაქსიმუმ ორი მარტივი რიცხვის ჯამისაგან არ შედგება.

ალგორითმი 4: ლუწ რიცხვთა შემოწმება (მესამე ვერსია)

```
1: n = 1;
2: do
3: n + +;
4: while(2n ≠ p1 + p2)        /* ეს ლუწი რიცხვი არ წარმოდგება ორი მარტივი რიცხვის ჯამის სახით */
```

ალგორითმი 5: შეჩერების დიაგონალური ალგორითმი D
მონაცემი: რაიმე ალგორითმი A

```
1: while(H(A(A))== „კი”)        /* შენიშვნა: აქ „A” ალგორითმის აღმწერი სიტყვაა (მაგ. მისი კოდი) */
```

ეს უკანასკნელი ამოცანა თანამედროვე მათემატიკის ერთ-ერთი უდიდესი ღია პრობლემაა (გოლდბახის ჰიპოთეზა) და, ცხადია, ჯერ-ჯერობით წინასწარ ვერ ვიტყვით, შეჩერდება თუ არა ზემოთ მოყვანილი ალგორითმი. ამრიგად, მოცემული ალგორითმის ანალიზით ზოგჯერ დაბეჯითებით შეიძლება იმის თქმა, შეჩერდება თუ არა იგი, ზოგჯერ კი -- არა.

განმარტება 1.5: მოცემულია რაიმე A ალგორითმი და მისი რაიმე X მონაცემი. საკითხს, შეჩერდება თუ არა A ალგორითმი X მონაცემზე ***შეჩერების ამოცანა*** ეწოდება.

კარგი იქნებოდა ისეთი ალგორითმის აღმოჩენა, რომელიც *ნებისმიერი* A ალგორითმისა და X მონაცემისთვის გვეტყოდა, ჩაიციკლება თუ არა $A(X)$. ამით მრავალი ღია პრობლემა გადაიჭრებოდა (მაგ. გოლდბახის ჰიპოთეზა). მაგრამ, სამწუხაროდ, ასეთი რამ შეუძლებელია, რაც შემდეგ თეორემაში მტკიცდება:

თეორემა 3.1: შეჩერების ამოცანა არაამოხსნადია
დამტკიცება: დავუშვათ საწინააღმდეგო: ვთქვათ, არსებობს ისეთი H ალგორითმი, რომელიც მონაცემად იღებს ნებისმიერი A ალგორითმისა და X მონაცემის აღმწერ სიტყვებს და გვეტყვის, შეჩერდება თუ არა $A(X)$. სხვა სიტყვებით რომ ვთქვათ,

$$H(A,X) = \begin{cases} \text{„კი”}, & A \text{ ალგორითმი შეჩერდება } X \text{ მონაცემზე;} \\ \text{„არა”}, & A \text{ ალგორითმი არ შეჩერდება } X \text{ მონაცემზე.} \end{cases}$$

ახლა კი შევქმნათ ისეთი ალგორითმი $D(X)$, რომელიც მონაცემად იღებს რაღაც X სიტყვას, რომელიც თავის თავად რაიმე ალგორითმის აღწერაა:

ალგორითმი 6: შეჩერების დიაგონალური ალგორითმი D
მონაცემი: რაიმე ალგორითმი A

```
1: while(H(A(A))== „კი”)        /* შენიშვნა: აქ „A” ალგორითმის აღმწერი სიტყვაა (მაგ. მისი კოდი) */
```

ესე იგი, D ალგორითმი მონაცემად მიიღებს რაიმე A ალგორითმის აღწერას და შეჩერდება მაშინ და მხოლოდ მაშინ, თუ A ალგორითმი მისსავე აღმწერი სიტყვის მონაცემზე არ შეჩერდება (აქ გასათვალისწინებელია ის ფაქტი, რომ თვით ალგორითმის აღწერაც რაღაც სიტყვაა, რომელიც მონაცემად შეიძლება ავიღოთ).
ახლა განვიხილოთ, თუ რა მოხდება, როდესაც D ალგორითმი თავის აღწერას მიიღებს მონაცემად. $D(D)$ ნიშნავს, რომ D ალგორითმი მისსავე აღმწერი სიტყვის მონაცემზე შეჩერდება მაშინ და მხოლოდ მაშინ, თუ D ალგორითმი მისსავე აღმწერი სიტყვის მონაცემზე არ შეჩერდება, ეს კი წინააღმდეგობაა!

რ.დ.გ.

ეს ფაქტი არ ნიშნავს აუცილებლად იმას, რომ რაღაცა ამოცანებისთვის ვერ ვიტყვით, ჩაიციკლება თუ არა მათი ალგორითმები. უსასრულოდ ბევრი ამოცანისათვის დაგვჭირდება ახალი მეთოდების ძიება, რომ ამ შეკითხვაზე პასუხი გავცეთ. შეჩერების ამოცანა ალგორითმულად ამოხსნადი რომ ყოფილიყო, მათემატიკური ლოგიკისა და ინფორმატიკის ერთ-ერთი უმნიშვნელოვანესი განხრა -- თეორემათა ავტომატური მტკიცება -- ადვილად გადაიჭრებოდა, ახლა კი საჭიროა თეორემათა კლასიფიკაცია და ყოველი ცალკეული კლასისათვის ავტომატური მტკიცებების მეთოდების ძიება. მსგავსი პრობლემები წამოიჭრება ასევე პროგრამული კოდის სისწორის ავტომატურ მტკიცებაშიც.

ძალიან მნიშვნელოვანია ასევე მეორე არაამოხსნადი ამოცანა, რომელიც „პოსტის შესაბამისობის პრობლემის” Post Correspondence Problem, მოკლედ PCP სახელითაა ცნობილი:

პოსტის შესაბამისობის პრობლემა (Post Correspondence Problem, PCP)
მოცემულია: $P = ((x_1, y_1), (x_2, y_2), ..., (x_n, y_n)), x_i, y_i \in \Sigma^+$ არაცარიელი სიტყვების წყვილთა მიმდევრობა, რომელსაც ჩვენ „ამოცანის პირობას", ან მოკლედ „პირობას" ვუწოდებთ.
$I = (i_1, ..., i_k), i_j \in \{1, 2, ..., n\}$ მიმდევრობას ეწოდება მოცემული პირობის ამონახსნი, თუ $x_{i_1} \circ x_{i_2} \circ \cdots \circ x_{i_n} = y_{i_1} \circ y_{i_2} \circ \cdots \circ y_{i_n}$.
თვალსაჩინოებისთვის P პრობლემა შეგვიძლია წარმოვიდგინოთ, როგორც დომინოს ქვები, რომელზედაც ერთ მხარეს x_i და მეორეზე კი y_i სიტყვა წერია, თან თითოეული ტიპის ქვა უსასრულოდ ბევრი შეიძლება იყოს (შესაძლებელია, რომ I ამოხსნაში $i_k = i_l$, თუ $k \neq l$.
მაშინ ამოცანა შემდეგში მდგომარეობს: შეიძლება თუ არა დომინოს ქვები ისე დავალაგოთ ერთმანეთის გვერდით, რომ ზედა ნაწილში წარმოქმნილი სიტყვა (სადაც მხოლოდ x_i ქვესიტყვები გვექნება) დაემთხვას ქვედა ნაწილში წარმოქმნილ სიტყვას?

მაგალითი 3.1: მოცემულია პრობლემა $P = ((1, 101), (10, 00), (011, 11))$, რაც გვაძლევს: $x_1 = 1, x_2 = 10, x_3 = 011$ და $y_1 = 101, y_2 = 00, y_3 = 11$. ამის ამონახსნი იქნება $I = (1, 3, 2, 3)$:
$x_1 \circ x_3 \circ x_2 \circ x_3 = 1 \circ 011 \circ 10 \circ 011 = 101 \circ 11 \circ 00 \circ 11 = 101110011 = 101 \circ 11 \circ 00 \circ 11 = y_1 \circ y_3 \circ y_2 \circ y_3$
თუ ამას წარმოვიდგენთ, როგორც დომინოს ქვების მიმდევრობას, მივიღებთ:

$$\boxed{\frac{1}{101}}\boxed{\frac{011}{11}}\boxed{\frac{10}{00}}\boxed{\frac{011}{11}}$$

ცხადია, რომ თუ ერთ ამოცანას რამოდენიმე ამოხსნა აქვს, მათი კომბინაციაც ამოხსნას გვაძლევს. ამიტომ შეიძლება წამოიჭრას შეკითხვა: თუ მოცემულია რაიმე ამოხსნა, შეიძლება თუ არა მისი სხვა ამოხსნების კომბინაციებად წარმოდგენა? წინა მაგალითში მოყვანილი ამოხსნა არ შეიძლება ასე დაიყოს. ანუ, როგორც ამბობენ, იგი **პრიმიტიულია**.
ზედა მაგალითის განხილვისას შეიძლება წარმოიშვას შთაბეჭდილება, რომ პოსტის პრობლემა სულაც არ არის რთული. ამიტომ განვიხილოთ შემდეგი
მაგალითი 3.2:
$P = ((001, 0), (01, 011), (01, 011), (10, 001))$; $x_1 = 001, x_2 = 01, x_3 = 01, x_4 = 10, y_1 = 0, y_2 = 011, y_3 = 101, y_4 = 001$.
მისი უმოკლესი ამოხსნაც კი 66 წყვილისაგან შედგება, რითაც ამ პრობლემის სირთულის პირველი შეფასება შეიძლება:

$$\begin{aligned} I = {} & (2,4,3,4,4,2,1,2,4,3,4,3,4,4,3,4,4,2,1,4,4,2,1,3,4,1,1,3,4,4,4,2,1,2,1,1,1,3,4,3,4,1,2,1,4,4,2,1,4,1, \\ & 1,3,4,1,1,3,1,1,3,1,2,1,4,1,1,3) \end{aligned}$$

სრული გადარჩევის მეთოდით შეიძლება შეიქმნას ალგორითმი, რომელიც პოსტის ამოცანის I ამოხსნას მოგვცემდა, თუ ასეთი არსებობს. ამოხახსნის არარსებობის შემთხვევაში ასეთი ალგორითმი ჩაიციკლება.

სავარჯიშო 1.22: დაწერეთ პროგრამა, რომელიც სრული გადარჩევით მოგვცემს PCP ამოცანის პასუხს ასეთის არსებობის შემთხვევაში. დაამტკიცეთ მისი სისწორე და დაითვალეთ ბიჯების რაოდენობა იმ დაშვებით, რომ მოცემული პირობისთვის ამონახსნი არსებობს.

უხეშად, ამოცანები ორ კლასად შეიძლება დაიყოს: ისეთები, რომლისთვისაც ალგორითმული ამოხსნა არ არსებობს (ანუ ამ ამოცანების გადაჭრა სასრულ დროში არ შეიძლება - მათი ყველა ალგორითმი მინიმუმ ერთ მონაცემზე მაინც ჩაიციკლება) და ისეთებად, რომლებისთვისაც ალგორითმული ამოხსნა არსებობს. ისეთ ამოცანებში, რომლებისთვისაც ალგორითმები არსებობს, გამოყოფენ ამოცანათა სირთულის რამოდენიმე კლასს. აღმოჩნდა, რომ ერთ-ერთ კლასში ისეთი ამოცანებია, რომელთათვისაც რეალურ (ანუ პოლინომიურ) დროში ამოხსნა არსებობს, ხოლო მეორეში - ისეთები, რომელთათვისაც რეალურ დროში ამოხსნა დღეისათვის ცნობილი არ არის. აქვე უნდა აღინიშნოს, რომ „ეფექტური", „სწრაფი", „რეალურ დროში" ამოხსნადი ალგორითმები იგივეა, რაც „პოლინომურ დროში ამოხსნადი" (იხ. პირველი სემესტრის ლექციათა კურსი).

მნიშვნელოვანია შემდეგი გარემოება: ფარგლითა და სახაზავით კონკრეტული გეომეტრიული ფიგურების აგების ამოცანა გადაუჭრელია მხოლოდ გარკვეული მეთოდების გამოყენებით -- თუ ჩვენ საკითხს ისე დავსვამთ, შეიძლება თუ არა იგივე ფიგურების რაიმე მეთოდით აგება, მაშინ პასუხი დადებითი იქნება. დანარჩენი აქ განხილული არაამოხსნადი ამოცანები კი ზოგადად გადაუჭრელია -- არ არსებობს რაიმე მეთოდი, რომელიც ამ შეკითხვებზე პასუხს სასრულ დროში გაგვცემდა. ეს კი იმას ნიშნავს, რომ ამოცანის ჩამოყალიბებისას მნიშვნელოვან როლს თვით რესურსების შერჩევა თამაშობს.

მნიშვნელოვანი შენიშვნა: ჩვენ აქ შეჩერების ამოცანის არაამოხსნადობა გარკვეული აპარატის, კერძოდ ჩვენი ფსევდოკოდის მეშვეობით დავამტკიცეთ, რაც არაამოხსნადობის მტკიცების გარკვეულ ინტუიციას იძლევა. მაგრამ არავის უთქვამს, რომ არ არსებობს რაიმე სხვა ენა ან მეთოდი, რითაც ეს ამოცანა სასრულ დროში ამოიხსნებოდა. ლოგიკურია შეკითხვა, რა აპარატია ისეთი „უნივერსალური", რომლით დამტკიცებული თეორემაც ნებისმიერ სხვა გამოთვლის საშუალებასაც მოიცავს? ამაზე ცალსახა პასუხი არ არსებობს, თუმცა საყოველთაოდ მიღებულია ე.წ. ***ჩერჩის თეზისი,*** რომელიც ცნობილმა ამერიკელმა მეცნიერმა ალონსო ჩერჩმა ჩამოაყალიბა, რომლის მიხედვითაც თუ რაიმე ამოცანა არ ამოიხსნება ე.წ. ტიურინგის მანქანით (რომელსაც ხშირად ასევე ტიურინგ-პოსტის მანქანასაც უწოდებენ), მაშინ იგი ვერანაირი სხვა მეთოდით ვერ გადაიჭრება სასრულ დროში. აქვე უნდა აღინიშნოს, რომ ტიურინგის მანქანა მარტივი გამომთვლელის მათემატიკური მოდელია, რომელიც ხშირად გამოიყენება თეორიულ ინფორმატიკაში, როგორც ზოგადი გამომთვლელი, მაგრამ მისი ფორმალურად ჩამოყალიბება და მასზე დაყრდნობით თეორემათა მტკიცება ჩვენი კურსის ფარგლებს ცდება - ჩვენ ალგორითმების ზოგადი განმარტებითა და ფსევდოკოდით შემოვიფარგლებით.

1.4 რეკურსიული და რეკურსიულად გადათვლადი ამოცანები

როგორც უკვე აღვნიშნეთ, გადაწყვეტილების ამოცანების დახმარებით ოპტიმიზაციის ამოცანის გადაჭრა შეიძლება. მაგრამ როგორ შეგვიძლია თვით გადაწყვეტილების ამოცანის ჩამოყალიბება? ამ შეკითხვაზე პასუხის გასაცემად განვიხილოთ ზოგადი მიდგომა: გადაწყვეტილების ამოცანაში გვაინტერესებს, აკმაყოფილებს თუ არა რაიმე A მონაცემი B თვისებას. მაგალითად, A მონაცემად შეიძლება განვიხილოთ რაიმე გრაფი და B თვისებად – ჰამილტონის ციკლი (ამ შემთხვევაში მივიღეთ HC), ან კონიუნქციური ნორმალური ფორმით ჩაწერილი ფუნქცია და თვისება, რომ ეს ფუნქცია ცვლადების რაიმე მნიშვნელობაზე იღებს პასუხს 1 (ამოცანა SAT) და მრავალი სხვა.

სავარჯიშო 1.23: ჩამოაყალიბეთ IS_k, $Clique_k$, TSP_k და VC_k გადაწყვეტილების ამოცანების მონაცემები და შესამოწმებელი თვისებები.

სიმარტივისთვის ჩვენ აქ და შემდგომში ამოცანასა და მისი ამონახსნების სიმრავლეს (ამოცანის ენას) გავაიგივებთ და უბრალოდ „ამოცანას" ან „პრობლემას" ვუწოდებთ.

განმარტება 1.6: თუ მოცემული L_A ამოცანისთვის არსებობს რაიმე ალგორითმი, რომელიც მოცემული x სიტყვისთვის შეჩერდება იმ შემთხვევაში, თუ $x \in L_A$ (და „კი" პასუხსაც მოგვცემს), მაგრამ **შეიძლება** არ შეჩერდეს, თუ $x \notin L_A$, მაშინ ასეთ ენას (ამოცანას) ***რეკურსიულად გადათვლადი*** ეწოდება. თუ იგივე ამოცანისთვის არსებობს ისეთი ალგორითმი, რომელიც ნებისმიერი სიტყვისთვის შეჩერდება და $x \in L_A$ შეკითხვაზე სწორ პასუხსაც გაგვცემს, მაშინ ასეთ ენას (ამოცანას) ***რეკურსიული*** ეწოდება.

სავარჯიშო 1.24: რა დამოკიდებულებაა რეკურსიულ ამოცანათა და რეკურსიულად გადათვლადი ამოცანების სიმრავლეებს შორის? პასუხი დაასაბუთეთ.

აღსანიშნავია, რომ ტერმინი „რეკურსიული" და „რეკურსიულად გადათვლადი" 1930ან წლებში განვითარებული რეკურსიული ფუნქციების თეორიიდან მოდის: ნებისმიერი *გამოთვლადი* ფუნქცია შეიძლება აღიწეროს რამოდენიმე მარტივი ფუნქციის რეკურსიული ჯაჭვის აგებით. მაგრამ არსებობს ისეთი ფუნქციებიც, რომელთა გამოთვლაც ამ მეთოდით არ შეიძლება ან გარკვეულ შემთხვევაში არ შეიძლება (როგორც ეს წინა განმარტებაშია აღნიშნული).

რეკურსიულად გადათვლადი ამოცანის ერთ-ერთი ყველაზე ცნობილი მაგალითია ზემოთ მოყვანილი PCP. მისი არაამოხსნადობის დამტკიცება ცნობილი არაამოხსნადი ამოცანის, კერძოდ კი შეჩერების პრობლემის მასზე „დაყვანით" მტკიცდება (ანუ PCP ამოცანის მეშვეობით შეჩერების ამოცანის გადაჭრის შესაძლებლობით - დაახლოებით ისე, როგორც გადაწყვეტილების ამოცანით ოპტიმიზაციის პრობლემას ვხსნიდით). დაყვანის არსსა და ტექნიკას ჩვენ შემდგომში დაწვრილებით განვიხილავთ.

სავარჯიშო 1.25: დაამტკიცეთ, რომ შემდეგი ამოცანები რეკურსიულად გადათვლადია (ჩათვალეთ, რომ ალგორითმის მონაცემი ორობით ანბანზეა მოცემული):

- მოცემული A ალგორითმისთვის განსაზღვრეთ, შეჩერდება თუ არა იგი $\mathbb{B}^n, n \in \mathbb{N}$ ენის ნებისმიერ მონაცემზე;
- მოცემული A ალგორითმისთვის განსაზღვრეთ, შეჩერდება თუ არა იგი $\mathbb{B}^n, n \in \mathbb{N}$ ენის რაიმე მონაცემზე;

- ორი A და B ალგორითმისთვის განსაზღვრეთ, არსებობს თუ არა $w \in \mathbb{B}^*$ მონაცემი, რომლისთვისაც $A(w) \neq B(w)$.

სავარჯიშო 1.26: შემდეგი გამონათქვამებიდან რომელია ჭეშმარიტი და რომელი - მცდარი? პასუხი დაასაბუთეთ

- თუ ამოცანა რეკურსიულად გადათვლადია, იგი ასევე რეკურსიულიცაა;
- თუ ამოცანა რეკურსიულია, იგი ასევე რეკურსიულად გადათვლადიცაა;
- K რეკურსიულად გადათვლადი ამოცანის ალგორითმი შეიძლება არ შეჩერდეს რაიმე $x \in L_K$ მონაცემზე;
- K რეკურსიულად გადათვლადი ამოცანის ალგორითმი შეიძლება არ შეჩერდეს რაიმე $x \notin L_K$ მონაცემზე;
- K რეკურსიულად გადათვლადი ამოცანის ალგორითმი შეიძლება შეჩერდეს რაიმე $x \notin L_K$ მონაცემზე;
- K რეკურსიული ამოცანის ალგორითმი შეიძლება შეჩერდეს რაიმე $x \notin L_K$ მონაცემზე;
- K რეკურსიული ამოცანის ალგორითმი შეიძლება არ შეჩერდეს რაიმე $x \notin L_K$ მონაცემზე.

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 siganeSi Zebnis algoriTmi

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, მნიშვნელობები - 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 7: 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
```

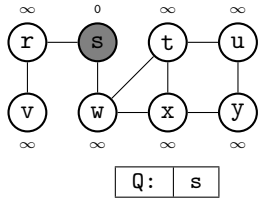
მშობლად ცხადდება u და მანძილად ფესვამდე - d[u]+1, ხოლო თავად ეს წვერო თავსდება Q რიგის ბოლოში. ამის შემდეგ u წვერო იღებება შავად (16 სტრიქონი).
სურ. 2.1-ზე მოცემულია BFS პროცედურის მუშაობა არაორიენტირებული გრაფისათვის და აგებულია სიგანეში ძებნის ხე. მუქად აღნიშნულია **ხის** წიბოები T (tree edges). წიბოები, 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 $\delta(s, v) \leq \delta(s, u) + 1$.

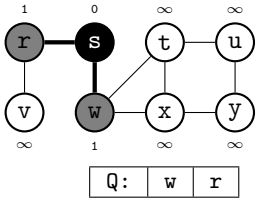
Proof. თუ u წვერო მიღწევადია s-დან, მაშინ მიღწევადია v-ც. ამ შემთხვევაში, უმოკლესი გზა s-დან v-დე არ შეიძლება იყოს იმაზე გრძელი, ვიდრე უმოკლესი გზა s-დან u-მდე, რომელსაც მოსდევს (u,v) წიბო. ასე, რომ დასამტკიცებელი უტოლობა სრულდება. ხოლო, თუ u წვერო არ არის მიღწევადი s-დან, მაშინ, $\delta(s, u) = \infty$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

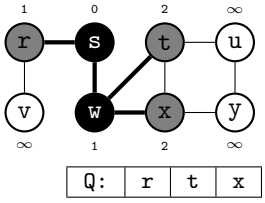
Proof. გამოვიყენოთ ინდუქცია ENQUEUE ოპერაციის რაოდენობის მიხედვით. ინდუქციის ჰიპოთეზა მდგომარეობს იმაში, რომ ყოველი $v \in V$ წვეროსთვის, სრულდება პირობა $d[v] \geq \delta(s, v)$.



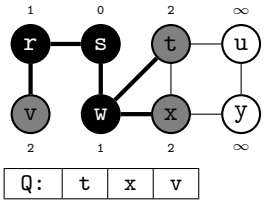
(a)



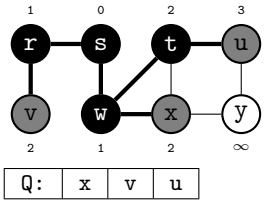
(b)



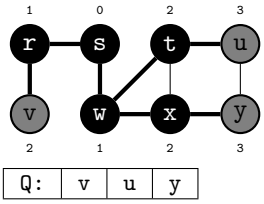
(c)



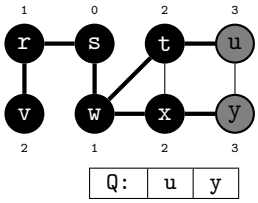
(d)



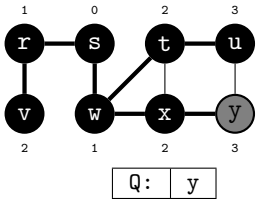
(e)



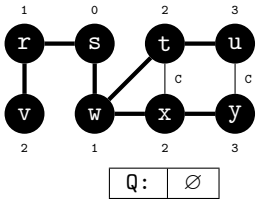
(f)



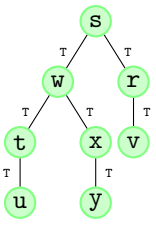
(g)



(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 წვეროს 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 წვეროს 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 8: 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 siRrmeSi Zebnis algoriTmi

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:

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

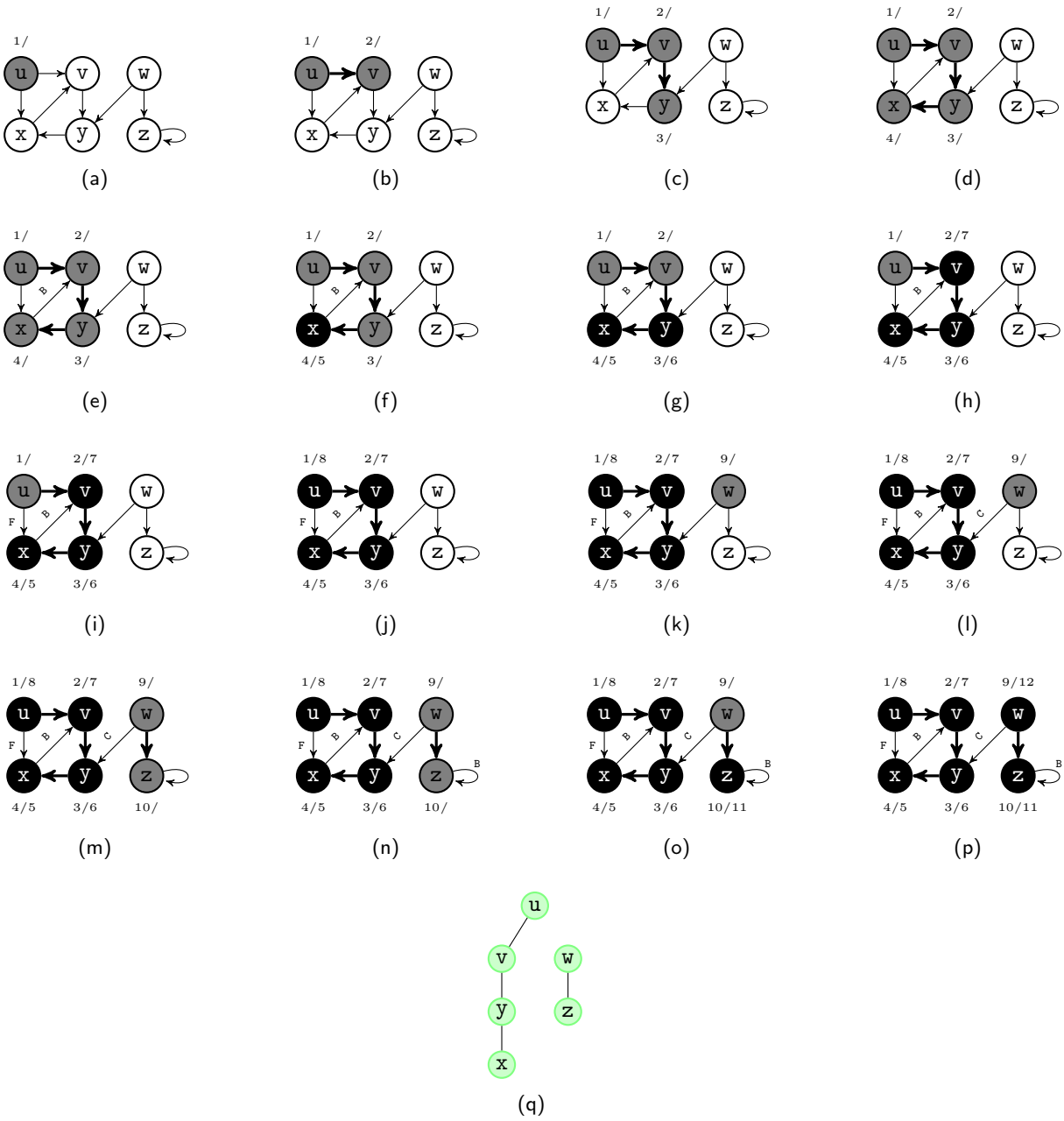
წინამორბედობის ქვეგრაფი წარმოადგენს სიღრმეში ძებნის ტყეს, რომელიც შეიძლება შედგებოდეს სიღრმეში ძებნის ხეებისაგან. E_π სიმრავლეში შემავალ წიბოებს უწოდებენ **ხის წიბოებს**.
სიღრმეში ძებნის ალგორითმიც იყენებს წვეროების შეფერვას. თავიდან ყველა წვერო თეთრია. წვეროს აღმოჩენის შემდეგ იგი ხდება რუხი. როცა წვერო 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 9: 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:

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

(a) (b)

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?

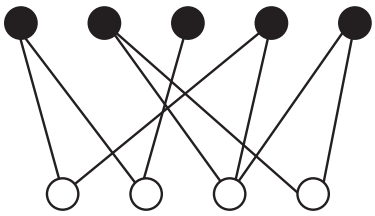
სავარჯიშო 2.5: შეიძლება თუ არა იგივე ამოცანის სიგანეში ძებნის მეთოდით გადაჭრა?

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 **??**.



ორად შეღებილი გრაფის მაგალითი

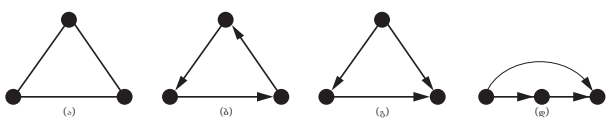
ორად შეღებილ გრაფს ზოგჯერ *ორად დაყოფილსაც* უწოდებენ. იმის დასადგენად, შეიძლება თუ არა მოცემული გრაფის ორად დაყოფა, შემდეგი სტრატეგიით შეიძლება მოქმედება:
ვირჩევთ ერთ-ერთ წვეროს, რომელსაც ვღებავთ რომელიმე ფრად (დავუშვათ, თეთრად). სიღრმეში ან სიგანეში ძებნით ახლად აღმოჩენილ წვეროს ვღებავთ მისი მშობლის (იმ წვეროსი, რომელთანაც არის დაკავშირებული) განსხვავებული ფერით (თეთრი ან შავი). შემდეგ *ყოველი აღმოუჩენელი* 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. **??**-ში მოყვანილი სამი გრაფის მაგალითი.



არამიმართული ციკლური, მიმართული ციკლური და მიმართული აციკლური გრაფები

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.
ასეთ სტრუქტურას *აციკლური მიმართული გრაფი* ეწოდება და მათი დახაზვა შეიძლება ისე, რომ ყველა წიბო *მარცხნიდან მარჯვნივ* იყოს მიმართული (მაგალითად, ნახ. **??**(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.

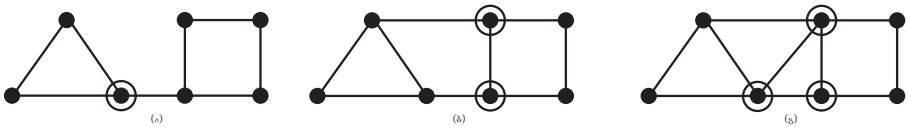
ტოპოლოგიურ დალაგებას დიდი მნიშვნელობა აქვს მთელ რიგ პრაქტიკულ ამოცანებში, მაგალითად ზემოთ ნახსენებ ცხრილის შედგენაში.

შიღრმეში ძებნის ალგორითმით ადვილად შეგვიძლია მიმართული გრაფის აციკლურობის დადგენა (რაც დამოკიდებულია იმაზე, შეგვხვდება 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.



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 wiboTa klasifikacia

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):

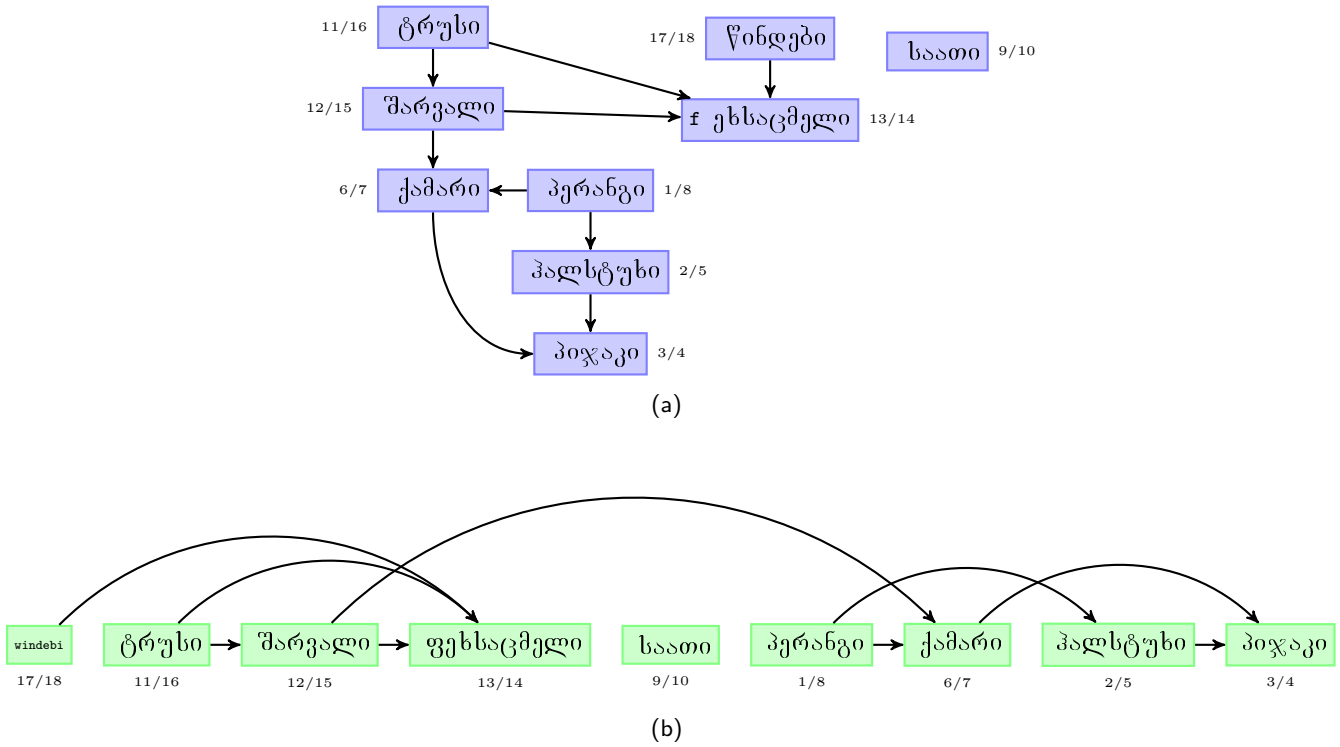
1. თეთრი ფერი - ხის წიბო
2. რუხი ფერი - უკუწიბო
3. შავი - პირდაპირი ან ჯვარედინი წიბო

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 topologiuri sortireba

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. 2.4ა-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. 2.4ბ-ზე.
წვეროების გვერდით მითითებულია მათი დამუშავების დაწყებისა და დამთავრების დროები. სურ. 2.4ბ-ში გრაფი ტოპოლოგიურად სორტირებულია. წვეროები დალაგებულია დამუშავების დამთავრების დროთა მიხედვით. ყველა წიბო მიმართულია მარცხნიდან მარჯვნივ.



ნახ. 2.4:

Algorithm 10: Topological Sort

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

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

შემდეგი ალგორითმი ტოპოლოგიურად ალაგებს ორიენტირებულ აციკლურ გრაფს.
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 Zlierad bmuli komponentebi

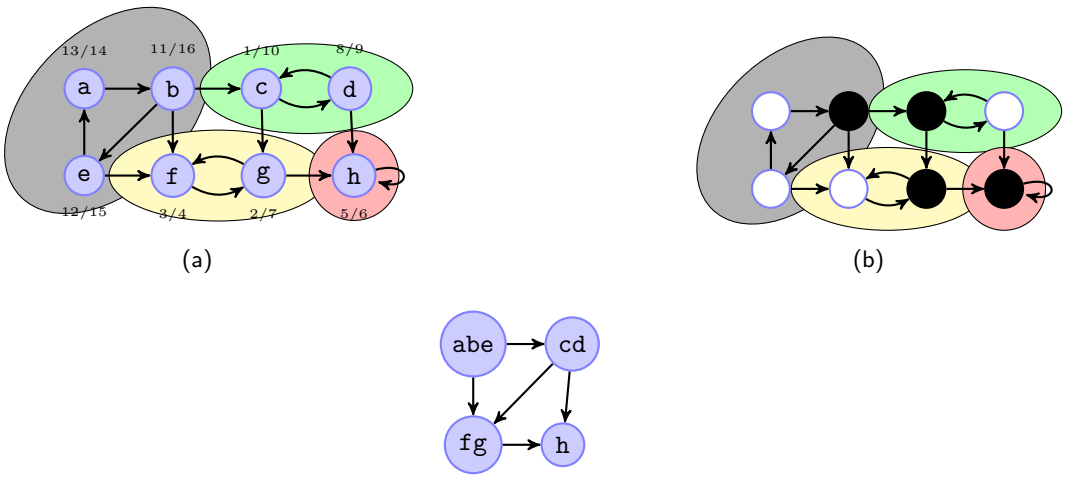
სიღრმეში ძებნის ალგორითმის გამოყენების კლასიკური მაგალითია გრაფის ძლიერად ბმულ კომპონენტებად დაშლა. ორიენტირებულ გრაფებზე მომუშავე მრავალი ალგორითმი იწყება გრაფში ძლიერად ბმული კომპონენტების მოძებნით. ამის შემდეგ ამოცანა იხსნება ცალკეული კომპონენტებისათვის, ხოლო შემდეგ ხდება კომბინირება ამ კომპონენტთა 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 11: 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
```



ნახ. 2.5:

სურ. 2.5ა-ზე რუხი ფონით აღნიშნულია G გრაფის ძლიერად ბმული კომპონენტები, ნაჩვენებია აგრეთვე სიღ-

რმეში ძებნის ტყე და დროის ჭდეები G^T გრაფისათვის. სურ. 2.5ბ-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. 2.5ც-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. 2.5ც.

კომპონენტების გრაფის ძირითადი თვისება მდგომარეობს იმაში, რომ ეს გრაფი წარმოადგენს აციკლურ ორიენტირებულ გრაფს, რაც გამომდინარეობს შემდეგი ლემიდან:

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 savarjiSoebi

1. სურ. 2.6 გრაფებისთვის განახორციელეთ სიგანეში ძებნა, ააგეთ სიგანეში ძებნის ხე.

ნახ. 2.6:

2. სურ. 2.6 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.

(ბ) 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. 2.7ა გრაფში დაალაგეთ წვეროები ტოპოლოგიური სორტირების ალგორითმით.

ნახ. 2.7:

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. 2.7ბ გრაფში.

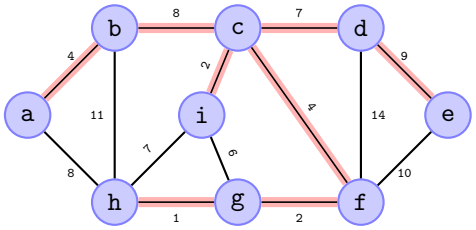
თავი 3

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 12: 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, მივითებთ 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) ("შეექმნათ 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.

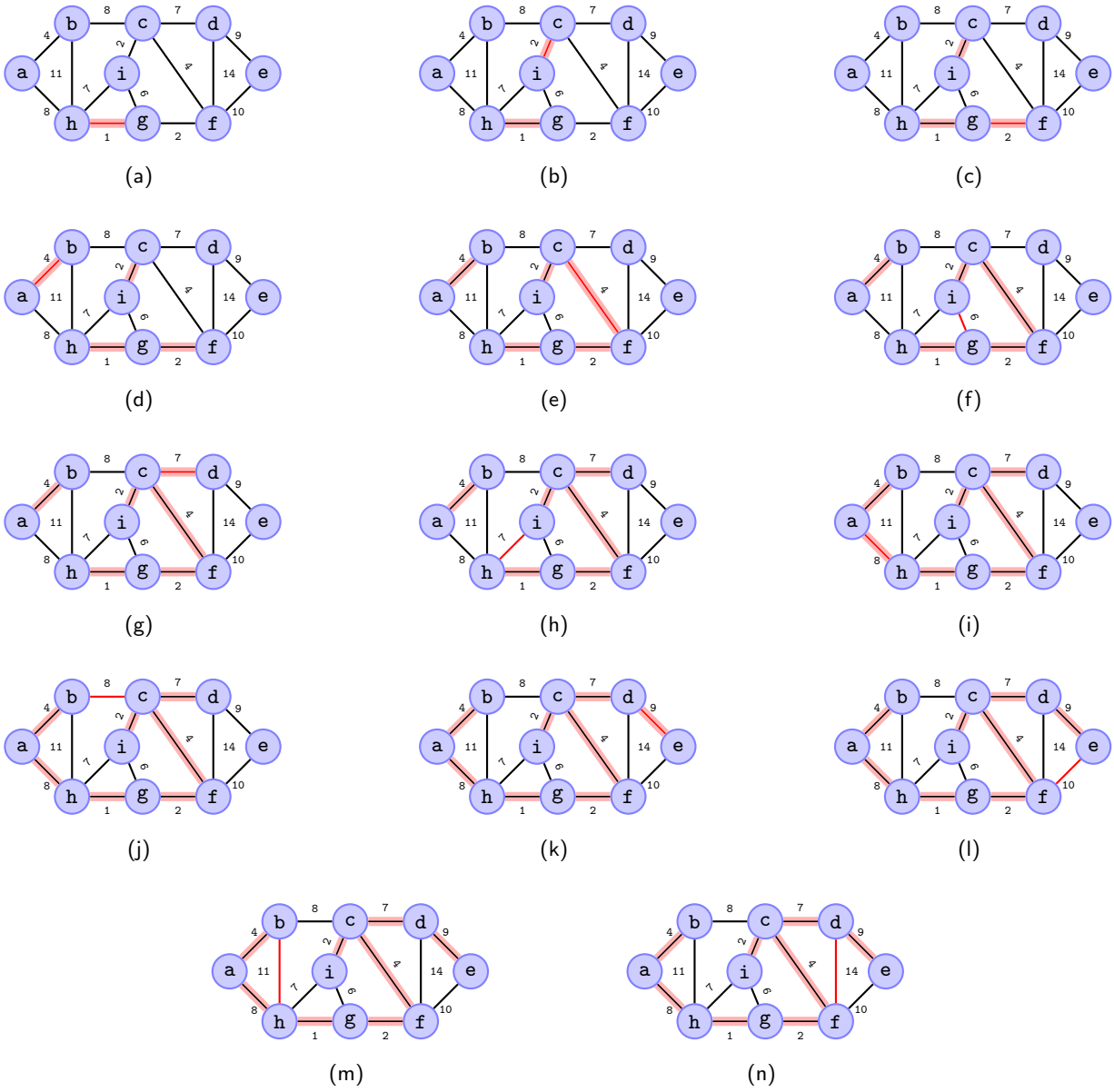
ზემოთ თქმულიდან გამომდინარე, გრაფის ორი u და v წვერო ეკუთვნის ერთ სიმრავლეს (ანუ კომპონენტს), როცა FIND-SET(u) = FIND-SET(v).

Algorithm 13: 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
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
⋮	⋮
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 14: 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 15: Minimum Spanning Tree - Prim

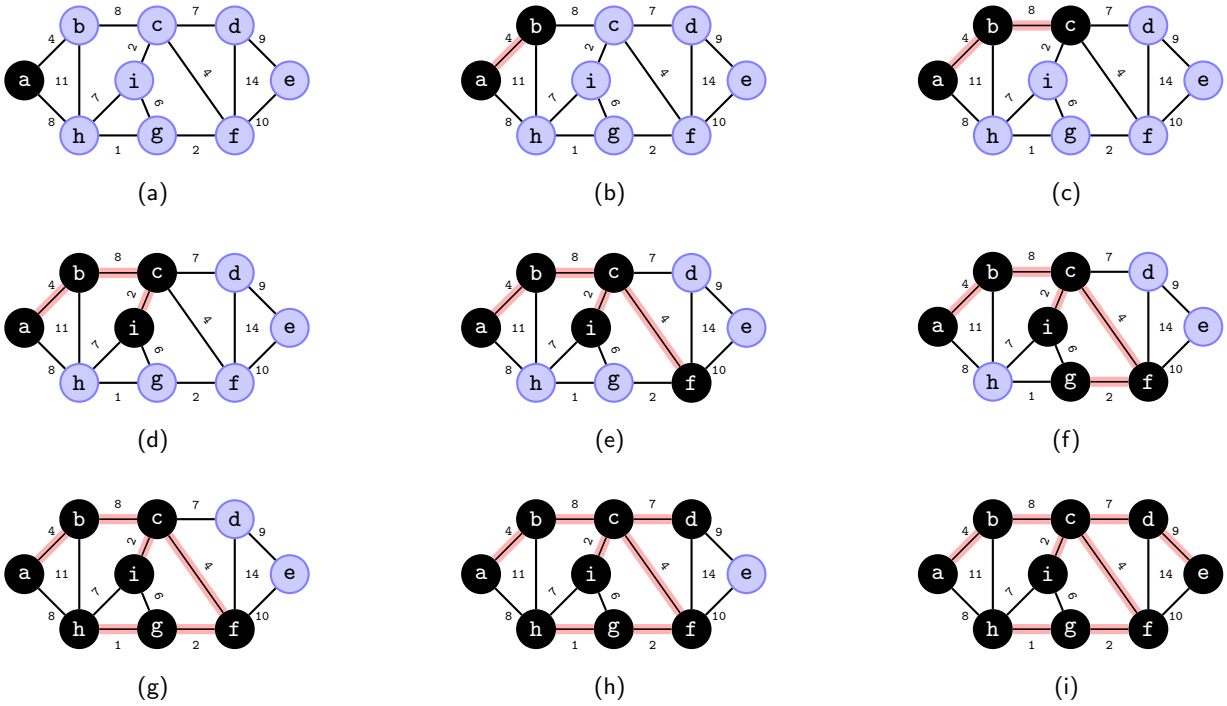
Input: ბმული არაორიენტირებული გრაფი $G = (V, E)$, წონის ფუნქცია $w : E \to R$ და საწყისი წვერო r
Output: გრაფის მინიმალური წონის დამფარავი ხე

```
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;

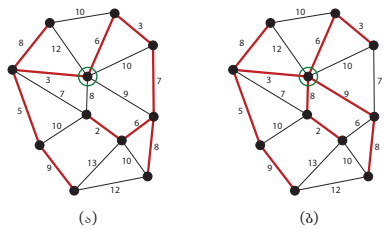
- ნალაპარაკევი ტექსტის ამოცნობის ერთ-ერთი უმთავრესი ამოცანა ერთნაირი ჟღერადობის სიტყვების (ომოფონების) განსხვავებაა. ასეთ სიტყვებზეა აგებული აკაკი წერეთლის ცნობილი ლექსი „აღმართ-აღმართ”:

  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.



(ა) (ბ)

მინიმალური დამფარავი ხე (ა) და შემოხაზული წვეროდან უმოკლესი მანძილის ხე (ბ)

სავარჯიშო 4.1: 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 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.

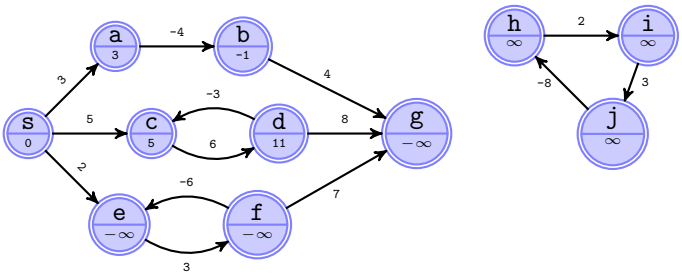
ლემა 4.1. (უმოკლესი გზის მონაკვეთები უმოკლესია). ვთქვათ, მოცემული გვაქვს ორიენტირებული წონადი $G = (V, E)$ გრაფი წონითი $w : E \to R$ ფუნქციით. თუ $p =< v_1, v_2, \ldots, v_k >$ უმოკლესი გზაა v_1-დან v_k-მდე და $1 \le i \le j \le k$, მაშინ $p_{ij} =< v_i, v_{i+1}, \ldots, v_j >$ წარმოადგენს უმოკლეს გზას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

Proof. თუ p გზას დავშლით შემადგენელ ნაწილებად, $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, 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 $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ 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. 4.1:

სურ. 4.1-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 16: 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 17: 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 $p =< v_0, v_1, \ldots, v_k > s = v_0$ წვეროდან v_k წვერომდე. ვთქვათ, შესრულდა INITIALIZE-SINGLE-SOURCE(G,s) პროცედურა, ხოლო შემდეგ, წიბოების შემდეგი თანმიმდევრობით რელაქსაცია:
$(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, 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 18: 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

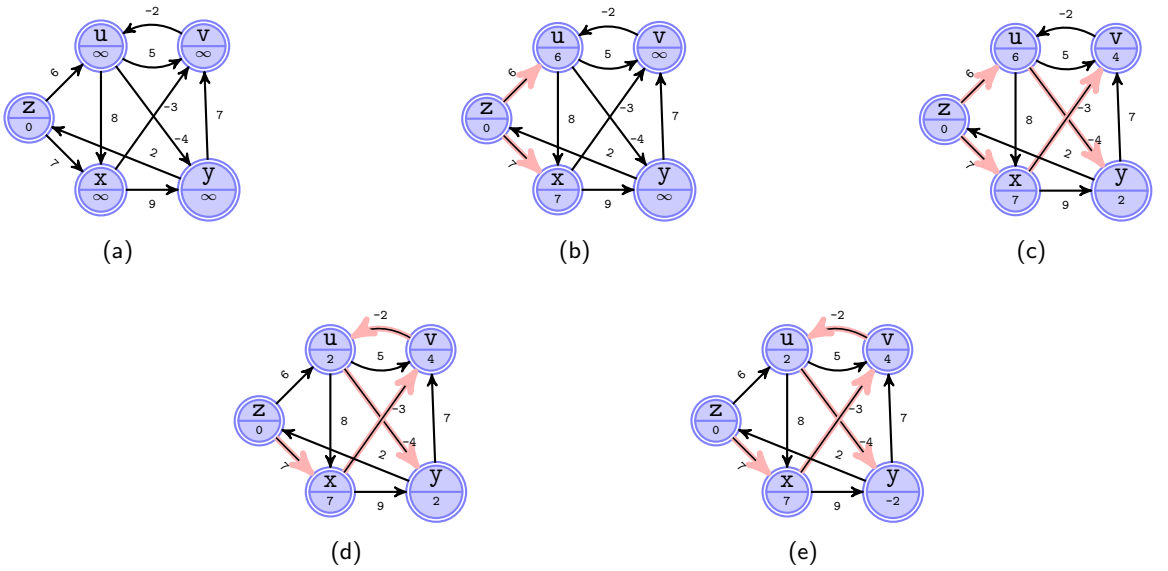
```
1 BELLMAN-FORD(G, w, s) :
2 |  INITIALIZE-SINGLE-SOURCE(G, s);
3 |  for i=1; i<|V|; i++ :
4 |  |  for ∀(u, v) ∈ E :
5 |  |  |  RELAX(u, v, w);
6 |  for ∀(u, v) ∈ E :
7 |  |  if d[v] > d[u] + w(u,v) :
8 |  |  |  return FALSE;
9 |  return TRUE;
```

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. 4.2-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. 4.2:

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 $k \leq |V|-1$. 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

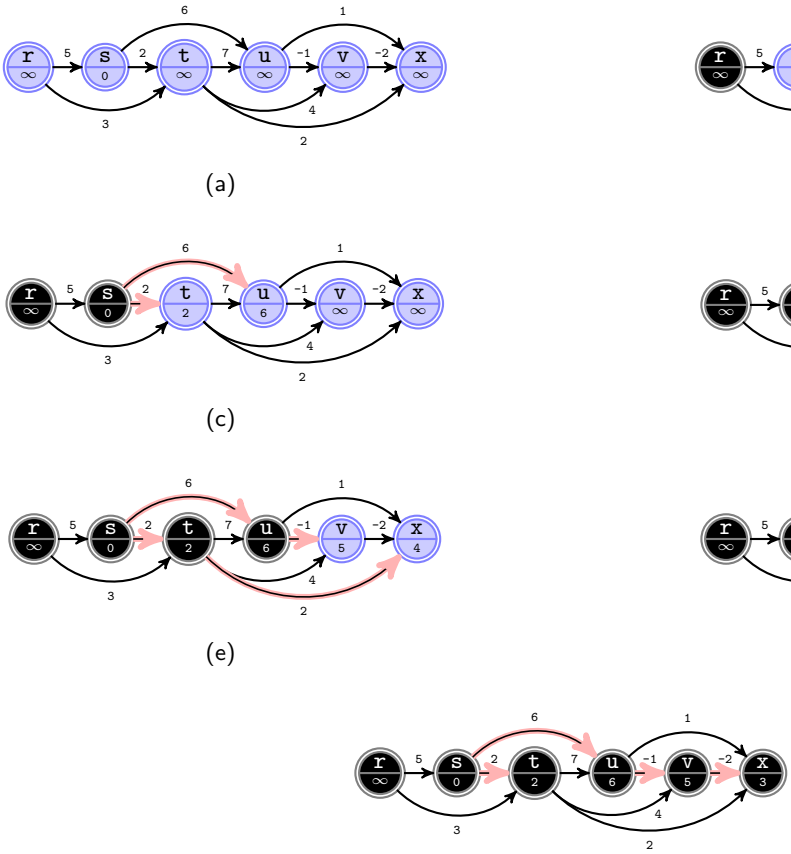
აციკლურ ორიენტირებულ $G = (V, E)$ გრაფში ერთი წვეროდან უმოკლესი გზების მოსაძებნად საჭიროა $\Theta(V+E)$ დრო, თუ წიბოების რელაქსაციას ჩავატარებთ ტოპოლოგიურად სორტირებული წვეროების მიხედვით. შევნიშნოთ, რომ აციკლურ ორიენტირებულ გრაფში უმოკლესი გზები ყოველთვის განსაზღვრულია, რადგან ციკლები (მათ შორის, უარყოფითწონიანიც) 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 19: 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. 4.3-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. 4.3:

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

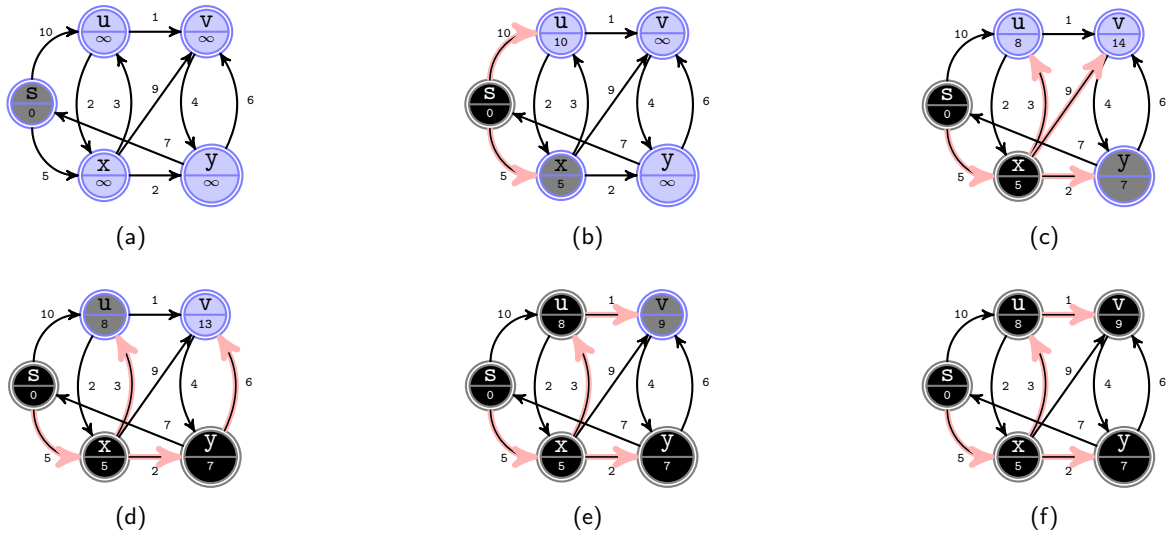
დეიქსტრას ალგორითმი პოულობს $G = (V, E)$ ორიენტირებული გრაფისათვის უმოკლეს გზებს საწყისი s წვეროდან ყველა დანარჩენ წვერომდე. აუცილებელია, რომ ყველა წიბოს წონა იყოს არაუარყოფითი $w(u, v) \geq 0$ ყოველი $(u, v) \in E$.

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 20: 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  <->სუბ*ცმტტ/მ/ნDIJKSTRA(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. 4.4:

დეიქსტრას ალგორითმის მუშაობის პროცესი აღწერილია სურ. 4.4-ზე. საწყისი წერტილია s. წვეროებში ჩაწერილია უმოკლესი გზების შეფასებები მოცემული მომენტისათვის. შავი ფერით აღნიშნულია წვეროები, რომლებიც S სიმრავლეს ეკუთვნიან. სხვა წვეროები დგანან $Q = V \backslash S$ პრიორიტეტებიან რიგში. რუხი ფერის წვეროები ციკლის მომდევნო იტერაციის დროს გამოდიან u წვეროს როლში. d და π თავიანთ საბოლოო მნიშვნელობებს ღებულობენ სურ. 4.4ფ-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 $Q = V \backslash S$. 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 $V \backslash S$ 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.

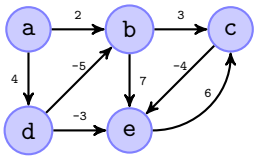
დეიქსტრას ალგორითმის მუშაობის დრო. ამ ალგორითმში გამოიყენება პრიორიტეტებიანი Q რიგი და სამი ოპერაცია (INSERT (სტრ. 4), EXTRACT-MIN (სტრ. 6), DECREASE-KEY (არაცხადად მონაწილეობს RELAX-ში) სტრ. 9) INSERT და EXTRACT-MIN პროცედურების გამოძახება ხდება ყოველი წვეროსთვის ერთხელ (წვეროების რაოდენობაა $|V|$). 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

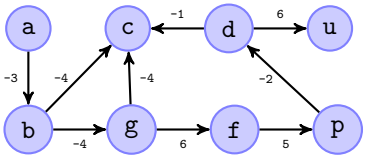
(ბელმან-ფორდის ალგორითმის მოდიფიკაცია) G გრაფის წვეროები გადავნომროთ ნებისმიერად და გრაფის წიბოთა E სიმრავლე გავყოთ ორ ნაწილად: E_f - წიბოები, რომლებიც მიმართულია ნაკლები ნომრის მქონე წვეროდან მეტი ნომრის მქონე წვეროსაკენ და E_b - წიბოები, რომლებიც მიმართულია მეტი ნომრის მქონე წვეროდან ნაკლები ნომრის 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

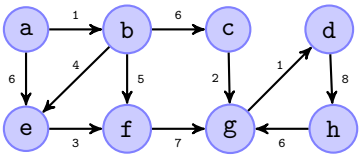
1. ბელმან-ფორდის ალგორითმით იპოვეთ უმოკლესი გზები a წვეროდან შემდეგ გრაფში:



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



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



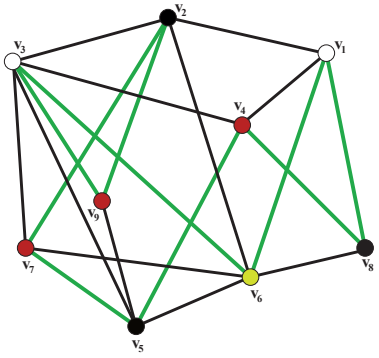
4. მოიყვანეთ უარყოფითი წონის მქონე წიბოს შემცველი ორიენტირებული გრაფის მაგალითი, რომლისთვისაც დეიქსტრას ალგორითმი იძლევა არასწორ შედეგს.

თავი 5

ამოცანათა სერტიფიცირება და მასზე დაფუძნებული სირთულის NP კლასი

5.1 სერტიფიცირება და სერტიფიკატი

განვიხილოთ შემდეგ ნახაზში მოცემული გრაფი. არსებობს თუ არა მასში ჰამილტონის ციკლი?



გრაფი ჰამილტონის ციკლისა და ოთხ ფრად შეღებვის სერტიფიკატით

ამ შეკითხვაზე პასუხის გაცემა ზოგადად რთულია. მაგრამ შესაძლებელია ვინმემ ჰამილტონის ციკლის არსებობა დაამტკიცოს შესაბამისი გზის წარმოდგენით: $(v_3, v_9, v_2, v_7, v_5, v_4, v_8, v_1, v_6, v_3)$.

ცხადია, რომ შესაძლებელია იმის გადამოწმება, ეს შემოთავაზებული გზა მართლაც ჰამილტონის ციკლია, თუ არა.

თუ იგივე გრაფზე დავსვამთ ოთხ ფრად შეღებვის ამოცანას, მაშინ შესაძლებელი იქნება ამ ამოცანის დადებითი პასუხის დამტკიცება კონკრეტული შეღებვის მოცემით (ნახ. **??**).

განმარტება 5.1: თუ რაიმე ამოცანისთვის შეიძლება შესაძლო ამონახსნის შემოთავაზება და ამ შემოთავაზებული სავარაუდო ამონახსნის გადამოწმების ალგორითმიც არსებობს, ამ შემოთავაზებულ ამონახსნს ***„სერტიფიკატი"***, ხოლო გადამოწმების ალგორითმს კი სერტიფიცირების პროცესი (ან, მოკლედ, ***სერტიფიცირება***) ეწოდება.

A ამოცანის სერტიფიცირების ალგორითმს აღვნიშნავთ როგორც V_A. ცხადია, რომ მისი მონაცემები იქნება თვით ამ A ამოცანის რაღაც ობიექტი X და შემოთავაზებული პასუხი y. მაგალითად, $V_{HC}(X, y)$, სადაც X რაღაც კონკრეტული გრაფია და y ამ გრაფის გზის სიტყვა, რომელიც მის ჰამილტონის ციკლს უნდა აღწერდეს.

სავარჯიშო 5.1: დაწერეთ HC, CN_k, VC_k და IS_k ამოცანების სერტიფიცირების ალგორითმები. რა იქნება ამ ალგორითმების მონაცემი და რა – პასუხი?

სავარჯიშო 5.2: რა იქნება V_{GI} და V_{SGI} მონაცემები?

5.2 NP კლასი, როგორც სწრაფად სერტიფიცირებად ამოცანათა სიმრავლე

ყველა ამოხსნადი ამოცანა, რომელიც აქამდე გვქონდა განხილული, პოლინომურ დროში, ანუ „სწრაფად" სერტიფიცირებადი იყო. ასეთ პრობლემებს ცალკე კლასში (სიმრავლეში) აერთიანებენ, რომელსაც NP ეწოდება:

$$\mathrm{NP} = \{A \mid A \text{ ამოცანა პოლინომურ დროში სერტიფიცირებადია}\}$$

ამოცანების (და, საერთოდ, გამოსაკვლევი ობიექტების) ერთ კლასში გაერთიანება იმითაა მოსახერხებელი, რომ მათი შემდგომი შესწავლა და აღწერა უფრო მოსახერხებელი იყოს. მეორე მნიშვნელოვანი კლასია P, რომელიც პოლინომურ დროში ამოხსნადი ამოცანებისგან შედგება:

$$\mathrm{P} = \{A \mid A \text{ ამოცანა პოლინომურ დროში ამოხსნადია}\}$$

ცხადია, რომ შესაძლებელია, რაიმე ამოცანა რომელიმე კლასში შედიოდეს (ანუ იყოს ამ კლასისთვის დამახასიათებელი თვისების მატარებელი), მაგრამ ეს არ იყოს ცნობილი. ასე, მაგალითად, რიცხვის სიმარტივის ტესტის ამოცანისთვის (რომელსაც ასევე *Prime* ეწოდება: მოცემული რიცხვისთვის განსაზღვრეთ, მარტივია იგი თუ არა) არ იყო ცნობილი სწრაფი (პოლინომური) ალგორითმი და მას სხვადასხვა ხერხით (მაგალითად შემთხვევითი ალგორითმებით) ხსნიდნენ, მაგრამ XXI საუკუნის დასაწყისში ინდოელმა მეცნიერებმა პოლინომური ალგორითმი გამოიგონეს, რითიც $Prime \in \mathrm{NP}$ დამტკიცდა.
ასევე არ არის ცნობილი, ამოიხსნება თუ არა VC_k ამოცანა პოლინომურ დროში, ამიტომაც მას ვერ მივაკუთვნებთ P კლასს, თუმცა არაა გამორიცხული, რომ ეს ასე იყოს.
ცხადია, რომ თუ ამოცანა ამოიხსნება პოლინომურ დროში, იგი სწრაფად სერტიფიცირებადიც იქნება, ანუ $\mathrm{P} \subset \mathrm{NP}$.

სავარჯიშო 5.3: დაამტკიცეთ, რომ $HC, CN_k, VC_k, IS_k, GI, SGI \in \mathrm{NP}$

სავარჯიშო 5.4: P=NP დაშვებით, რა თვისება ექნება ნებისმიერ სწრაფად სერტიფიცირებად ამოცანას?

სავარჯიშო 5.5: დაამტკიცეთ $\mathrm{P} \subset \mathrm{NP}$.

ამ სავარჯიშოთი მივუახლოვდით თანამედროვე მეცნიერების ერთ-ერთ უდიდეს დია საკითხს: რადგან $\mathrm{P} \subset \mathrm{NP}$, არ არის გამორიცხული, რომ $\mathrm{P} = \mathrm{NP}$, მაგრამ ასევე შესაძლებელია, რომ $\mathrm{P} \neq \mathrm{NP}$ (თეორიულად ისიცაა შესაძლებელი, რომ ორივე შემთხვევა ჭეშმარიტი იყოს, ანუ ორი ერთმანეთისგან დამოუკიდებელი თეორია არსებობდეს, ერთში პირველი შემთხვევა იყოს ჭეშმარიტი, მეორეში კი მისი უარყოფა, მაგრამ ეს საკითხი ჩვენი კურსის ფარგლებს ცდება).
რომელია აქედან ჭეშმარიტი, ეს ჯერ-ჯერობით უცნობია, თუმცა ამ საკითხის გადაჭრაზე ძალიან ბევრი თეორიული და პრაქტიკული შედეგია დამოკიდებული.

თავი 6

NP-სრული და NP-რთული ამოცანები

6.1 ამოცანების ერთმანეთზე დაყვანის პრინციპი

დავუშვათ, მოცემული გვაქვს რაიმე A ამოცანა, რომლის ალგორითმიც ჩვენთვის ცნობილი არ არის. თუ არსებობს ისეთი B ამოცანა ჩვენთვის ცნობილი ალგორითმით და ისეთი ალგორითმი R_{AB}, რომლითაც A ამოცანის ნებისმიერ X მონაცემს ისე გარდავქმნით B ამოცანის ისეთ X' მონაცემად, რომ ამ უკანასკნელმა პირველი ამოცანის ამოხსნა მოგვცეს (ანუ $A(X) = B(X')$), მაშინ ამბობენ, რომ A ამოცანა B ამოცანაზე დაიყვანება (იხ. შემდეგი დიაგრამა).

$$\begin{array}{ccc} A(X) & \stackrel{?}{=} & Y \\ {\scriptstyle R_{AB}}\big\downarrow & & \\ B(X') & = & Y \end{array}$$

ცხადია, რომ R_{AB} ალგორითმიც (რომელსაც ასევე A ამოცანის B ამოცანაზე ***დაყვანის*** პროცესსაც უწოდებენ) სწრაფი უნდა იყოს, რომ ასეთ ოპერაციას რაიმე აზრი ქონდეს.
ასეთ შემთხვევაში ამბობენ, რომ B ამოცანა „არანაკლები სირთულისაა", ვიდრე A და ეს ცხადიცაა: B ამოცანის ამოხსნა უფრო „მარტივი" რომ ყოფილიყო, მაშინ საწყის ამოცანას სწრაფად („მარტივად") დავიყვანდით მასზე, ამოვხსნიდით და საერთო ჯამში A ამოცანის ამოხსნაც მარტივი იქნებოდა.
ამ შემთხვევაში წერენ: $A \leq_P B$ და ამბობენ, რომ A ამოცანა B ამოცანაზე დაიყვანება (პოლინომურ დროში).

შენიშვნა: აქ ნახმარი ტერმინები „მარტივი" და „რთული" მკაცრად შემდგომში იქნება განმარტებული, ახლა კი ინტუიციის გასანვითარებლად უფრო ზოგადი აღწერით შემოვიფარგლებით.

ამ იდეების უკეთ გასაგებად განვიხილოთ რამოდენიმე მაგალითი.

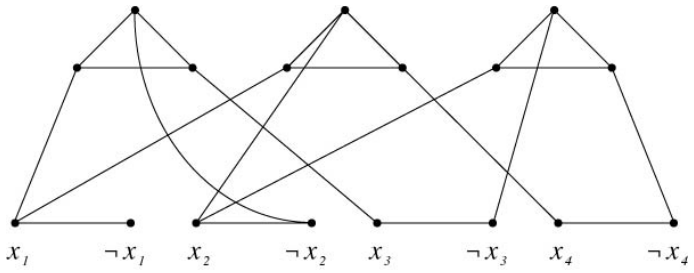
- IS_k და $Clique_k$: თუ მოცემულია იზოლირებული სიმრავლის ამოცანა, მისი გრაფის გარდაქმნა შეიძლება შემდეგნაირად: კვანძების სიმრავლე იგივე რჩება. თუ საწყის გრაფში ორი წვერო წიბოთი შეერთებული არაა, გარდაქმნილ გრაფში ეს ორი წვერო უნდა შეერთდეს და პირიქით: თუ საწყის გრაფში ორი წვერო წიბოთია შეერთებული, მაშინ გარდაქმნილ გრაფში მათ შორის წიბო არ იარსებებს. ადვილი საჩვენებელია, რომ ასეთ გარდაქმნილ გრაფში სრული სიმრავლის ამოცანა პასუხს გაგვცემს იზოლირებული სიმრავლის ამოცანაზე, ანუ პირველი ამოცანის მეორეზე დაყვანა შეიძლება.

- VC_k და $3SAT$: (ეს ბოლო ამოცანა იგივეა, რაც SAT, მხოლოდ ყოველ ფრჩხილში ზუსტად სამი ცვლადია წარმოდგენილი)

  დასაწყისისთვის განვიხილოთ კერძო მაგალითი: მოცემულია ფუნქცია, რომელიც ჩაწერილია კონიუნქციური ნორმალური ფორმით და შედგება 4 ცვლადისა და 3 დიზიუნქციური ჩანაწერისაგან (ანუ სამი ფრჩხილისაგან, რომელშიც ასევე სამი ცვლადი შედის):

$$F(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2 \vee x_3)(x_1 \vee x2 \vee x4)(x_2 \vee \neg x_3 \vee \neg x_4).$$

ყოველ ცვლადსა და მის უარყოფას შევუსაბამოთ ერთმანეთთან წიბოთი შეერთებული თითო კვანძი გრაფში, ხოლო ყოველ დიზიუნქციურ ჩანაწერს შევუსაბამოთ სამი ერთმანეთთან წიბოთი შეერთებული კვანძი (იხ. ნახ. **??**). თუ დიზიუნქციურ ჩანაწერში (ფრჩხილში) შედის რომელიღაცა ცვლადი x_i ან $\neg x_i$, ამ ცვლადის (ან მისი უარყოფის) შესაბამისი კვანძი ჩანაწერის ერთ-ერთ კვანძს უერთდება წიბოთი.



რედუქციის სქემა

ადვილი საჩვენებელია, რომ k ცვლადიანი და c დიზიუნქციურ ჩანაწერიანი მოცემული ფუნქციის ასეთი სახით გარდაქმნის შედეგად მიღებული გრაფი გადაიფარება $k + 2c$ კვანძით მაშინ და მხოლოდ მაშინ, თუ ეს ფუნქცია შეიძლება ჭეშმარიტი გახდეს ცვლადების გარკვეული მნიშვნელობებისათვის (თუ არსებობს ცვლადების ასეთი მნიშვნელობები, მაშინ უნდა შეირჩეს მათი შესაბამისი კვანძები).

ზოგადად, ყოველი ცვლადისთვის შევქმნით ორ კვანძს, რომელიც წიბოთია ერთმანეთთან დაკავშირებული; ასევე ყოველი ფრჩხილისთვის შევქმნით სამ კვანძს და მათაც შევაერთებთ (მივიღებთ იმდენ სამკუთხედს, რამდენი ფრჩხილიც გვაქვს, თანაც თითო კვანძს ფრჩხილის თითო ცვლადი შეესაბამება). ბოლოს, თუ ფრჩხილში გვხვდება რაიმე ცვლადი ან მისი უარყოფა, შესაბამისი ცვლადის ან მისი უარყოფის შესაბამისი წვერო ამ ფრჩხილის შესაბამისი სამკუთხედის რომელიმე წვეროს უკავშირდება. ასე შექმნილი გრაფი უნდა გადაეცეს გადაფარვის VC_m ამოცანას. სადაც $m = k + 2c$. აქ k ცვლადების, c კი ფრჩხილების რაოდენობაა.

სავარჯიშო 6.1: აჩვენეთ, რომ ზემოთ მოყვანილი გრაფი გადაიფარება $k + 2c$ წვეროთი მაშინ და მხოლოდ მაშინ, თუ შესაბამისი ფუნქცია არაა ნულოვანი.

სავარჯიშო 6.2: დაწერეთ პროგრამა, რომელიც $3SAT$ შესაბამისად მოცემული დიზიუნქციური ფორმიდან ზემოთ აღნიშნულ გრაფს გამოითვლის (და შეძლებისამებრ დახაზავს).

- CN_k და IS_m:
  სავარჯიშო 6.3: აჩვენეთ, რომ $CN_k \leq_P IS_m$

თუ $A \leq_P B$ და $B \leq_P A$, მაშინ ამბობენ, რომ ორივე ამოცანა ერთი და იმავე სირთულისაა და წერენ $A =_P B$.

სავარჯიშო 6.4: აჩვენეთ, რომ $VC_k = CN_p$ და $VC_k = IS_p$ (გაითვალისწინეთ, რომ შესაძლებელია $k \neq p$).

6.2 NP რთული და NP სრული ამოცანები

განმარტება 6.1: თუ ამოცანათა რაიმე კლასიდან (სიმრავლიდან) *ყველა* ამოცანა რაიმე S ამოცანაზე დაიყვანება, მას ამ კლასის მიმართ ***რთული*** ეწოდება. ბუნებრივია, რადგან ჩვენი განხილვის მთავარი ობიექტი NP კლასია, ამიტომაც პირველ რიგში NP-რთულ ამოცანებს განვიხილავთ.
NPრთული ამოცანა ისეთიც შეიძლება იყოს, რომელიც თვითონ ეკუთვნის ამ კლასს. ამ შემთხვევაში მას NP-***სრული*** ეწოდება.

ბუნებრივია შეკითხვა: არსებობს კი ისეთი ამოცანა, რომელზედაც NPკლასის *ნებისმიერი* ამოცანა დაიყვანება? ასეთი ამოცანების არსებობა ძალიან საინტერესო შედეგების გამომწვევი იქნებოდა, რადგან მხოლოდ NP-სრული S ამოცანის ამოხსნით ნებისმიერი სხვა $A \in$ NP ამოცანის ამოხსნის გზას ვიპოვნიდით: A ამოცანას დავიყვანდით S ამოცანაზე და მისი ამოხსნით საწყისი ამოცანის პასუხსაც მივიღებდით (რა თქმა უნდა, თვითონ დაყვანის ალგორითმის პოვნაც საჭიროა, მაგრამ ეს უფრო ადვილია, მით უმეტეს, რომ ასეთის არსებობა გარანტირებულია).

აღმოჩნდა, რომ NP-სრული და, შესაბამისად, NP-რთული ამოცანაც ძალიან ბევრი არსებობს.

იმის საჩვენებლად, რომ რაიმე A ამოცანა NP-სრულია, საკმარისია იმის ჩვენება, რომ რაიმე უკვე ცნობილი NPსრული ამოცანა S მასზე დაიყვანება: თუ ვიცით, რომ ნებისმიერი $X \in \text{NP}$ ამოცანისთვის $X \leq_P S$ და $S \leq_P A$, მაშინ $X \leq_p A$.

სავარჯიშო 6.5: დაამტკიცეთ $A \leq_P B \leq_P C \Rightarrow A \leq_P C$ (მინიშნება: გაიაზრეთ, რას ნიშნავს $X \leq_P Y$ და შემდეგ ერთმანეთის მიყოლებით ანალოგიური ალგორითმების ჩატარება რამდენად სწრაფ ალგორითმს მოგვცემს).
შენიშვნა: ამ სავარჯიშოს ერთი საინტერესო „მახე" აქვს, რომელიც აუცილებლად უნდა გაითვალისწინოთ: თუ ვინმე ეცდება A ამოცანიდან B ამოცანაზე დაყვანის R_{AB} ალგორითმისა და, შესაბამისად, R_{BC} დაყვანის ალგორითმების ჩვეულებრივი „კომპოზიციით" A ამოცანიდან C ამოცანაზე დაყვანის R_{AC} ალგორითმის აგებას, შეიძლება შემდეგ პრობლემას წააწყდეს: $x \in A$ მონაცემის გარდაქმნისას მივიღოთ გაცილებით უფრო გრძელი მონაცემი, ვიდრე $|x|$ (ანუ A ამოცანიდან C ამოცანაზე დაყვანის ალგორითმის მონაცემის სიგრძე), რაც ცალკე R_{BC} ალგორითმის მუშაობის დროზე არ იმოქმედებდა (რადგან დროის ზედა ზღვარი ამ გრძელი მონაცემის სიგრძესთან შედარებით დიდი არ იქნებოდა), მაგრამ საწყისი x მონაცემის სიგრძესთან შედარებით კომპოზიცია ცუდ შედეგს მოიტანს.

როგორც ზემოთ ნათქვამიდან და სავარჯიშოებიდენ ჩანს, შედარებით ადვილი უნდა იყოს ერთი ამოცანის მეორეზე დაყვანის პროცესის პოვნა (თუმცა ჩვენ აქ შედარებით მარტივ მაგალითებს განვიხილავდით: ეს პროცესი უფრო რთულიც შეიძლება იყოს, მაგრამ ძალიან რთული არაა). გაცილებით უფრო რთულია *პირველი* NP-სრული ამოცანის პოვნა, როდესაც, არაფორმალურად რომ ვთქვათ, პირველი ხელმოსაჭიდი არ გვაქვს და უნდა დამტკიცდეს, რომ ყველა ამოცანა მასზე შეიძლება დავიყვანოთ.
NPრთული ამოცანების თეორია 1970ანი წლების დასაწყისში განვითარდა ამერიკელი მეცნიერის სტივენ კუკის (Stephen Cook) და, მისგან დამოუკიდებლად, მოსკოველი მეცნიერის ლეონიდ ლევინის შრომებში – ორივემ ასეთი სრული პრობლემის არსებობა დაამტკიცა, თუმცა სხვადასხვა გზით. სამეცნიერო ლიტერატურაში ეს შედეგი კუკ-ლევინის თეორემის სახელითაა ცნობილი.
კუკის თეორემა გვეუბნება, რომ SAT ამოცანა NP-სრულია. ამ შედეგის პრაქტიკული მნიშვნელობა გამოჩნდა ამერიკელი მეცნიერის რიჩარდ კარპის (Richard Karp) 1972 წელს გამოქვეყნებულ ნაშრომში, სადაც მან 21 ამოცანის NPსრულობა დაამტკიცა: იქიდან გამომდინარე, რომ SAT NPსრულია, მან ეს ამოცანა დაიყვანა $3SAT$ პრობლემაზე, შემდეგ ეს უკანასკნელი VC ამოცანაზე, შემდეგ ეს IS-ზე და ა.შ. ამდაგვარი ჯაჭვით ყველა ამ ამოცანის NPსრულობა დამტკიცდა. დღეისათვის ამ კლასის ათასობით ამოცანა არსებობს, რომელთა თეორიულ და პრაქტიკულ მნიშვნელობას ჩვენ შემდგომში განვიხილავთ.

მართალია, პირველი კონკრეტული NPსრული ამოცანა არის SAT, სიმარტივისთვის ჩვენ ე.წ. $CirquitSAT$ (ან, როგორც ზოგიერთი ავტორი მოკლედ აღნიშნავს, $CSAT$) ამოცანის NPსრულობას დავამტკიცებთ და აქედან გამომდინარე ავაგებთ ჯაჭვს $CSAT \leq_P SAT$ ჩვენებით.
თვით $CSAT$ არის იგივე SAT ამოცანის ანალოგი, რომელიც სქემებზეა გადატანილი: მოცემულია $f : \mathbb{B}^n \to \mathbb{B}$ ფუნქციის სქემა C. არსებობს თუ არა მისი ისეთი მონაცემი, რომ C იძლეოდეს პასუხს 1 ?

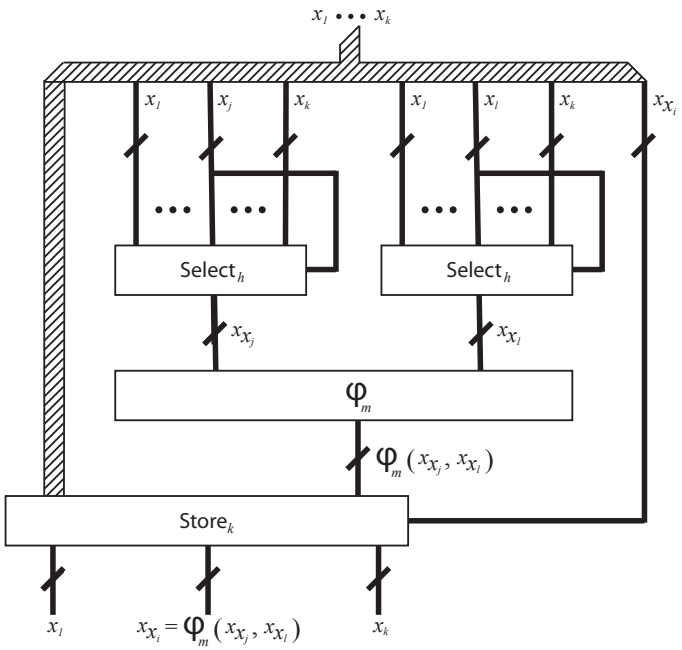
თეორემა 2.1: $CSAT$ ამოცანა NP-სრულია

დამტკიცება: დავუშვათ, მოცემულია რაიმე $A \in \text{NP}$ ამოცანა. მისთვის უნდა არსებობდეს სერტიფიცირების ალგორითმი $V_A(X, y)$, რომელიც A ამოცანის X ობიექტისთვის y შემოთავაზებულ პასუხს პოლინომურ დროში გადაამოწმებს (მაგალითად, A შეიძლება იყოს ჰამილტონის ციკლის ამოცანა, X რაიმე კონკრეტული გრაფი და y ამ გრაფის რაიმე გზა). ზოგადობის შეუზღუდავად დავუშვათ, რომ X და y მონაცემები ორობით ანბანშია ჩაწერილი.
თუ ვაჩვენებთ, რომ პოლინომურ დროში შესაძლებელია $V_A(X, y)$ ალგორითმის შესაბამისი სქემის აგება (ანუ ისეთი CV_A სქემის, რომელიც X და y მონაცემების შესაბამის შემავალ სიგნალებზე იგივე პასუხს მოგვცემს, როგორც $V_A(X, y)$, ყველაფერი დამტკიცებული იქნება: თუ მოცემული გვექნება A ამოცანის რაიმე X' ობიექტი და უნდა დავადგინოთ, გვაძლევს თუ არა $A(X')$ პასუხს „კი", შესაძლებელი იქნება CV_A სქემაში X' შესაბამისი სიგნალების დაფიქსირება, შემდეგ ამის გათვალისწინებით ამ სქემის გამარტივება (რაც წინა სემესტრში გვქონდა), რითაც მივიღებთ სქემას SCV_A, სადაც CV_A სქემისგან განსხვავებით გვექნება მხოლოდ y ელემენტების შესაბამისი მონაცემების შემომავალი სიგნალები. ასე რომ, თუ $CSAT(SCV_A)$ ამოცანა გვეტყვის პასუხს „კი", მაშინ უნდა არსებობდეს ისეთი y მონაცემი, რომლისთვისაც $V_A(X', y)$=„კი", რაც იმას ნიშნავს, რომ მოცემული $A(X')$=„კი". და თუ $CSAT(SCV_A)$ ამოცანა გვეტყვის პასუხს „არა", მაშინ **არ** უნდა არსებობდეს ისეთი y მონაცემი, რომლისთვისაც $V_A(X', y)$=„კი", ანუ $A(X')$=„არა".

ზემოთ ნათქვამიდან გამომდინარე, დასამტკიცებელია შემდეგი ლემა:

ლემა 2.1: ნებისმიერი R ალგორითმიდან, რომლის დროის ფუნქციაა $f_R(n)$, $O(f_R(n)^{const})$ დროში შეიძლება აიგოს შესაბამისი CR სქემა.

დამტკიცება: აქ ჩვენ არა მკაცრ დამტკიცებას, არამედ ძირითად იდეას მოვიყვანთ. ყოველი ალგორითმი შეგვიძლია განვიხილოთ, როგორც რაღაც $x_1, ..., x_k$ ცვლადების მიმდევრობა, რომელზედაც ყოველ ბიჯზე $x_i = \phi(y_j, y_l)$ ოპერაცია ტარდება, სადაც y_j, y_l ან ცვლადები, ან მუდმივებია (მაგ. $x_3 = x_{507} + x_3$, ან $x_9 = x_{71} \cdot 12$ და ა.შ.).
ყოველი x_i ცვლადისთვის, რომელიც შეგვხვდება ჩვენს ალგორითმში, განვათავსოთ (თითო ცვლადისთვის შესაბამისი ბიტების რაოდენობის) ვერტიკალური მავთულები.
ცხადია, რომ ყოველი ϕ ოპერაციისთვის შესაძლებელია ფიქსირებული სქემის შექმნა (იმ პირობით, თუ მონაცემების ბიტების რაოდენობა წინასწარაა დაფიქსირებული).
თუ გვაქვს გამოსახულება $x_{x_i} = \phi_m(x_{x_j}, x_{x_l})$, სადაც ϕ რაიმე კონკრეტული ოპერაციაა, მისი რეალიზაცია შეიძლება ქვემოთ მოყვანილი სქემით.

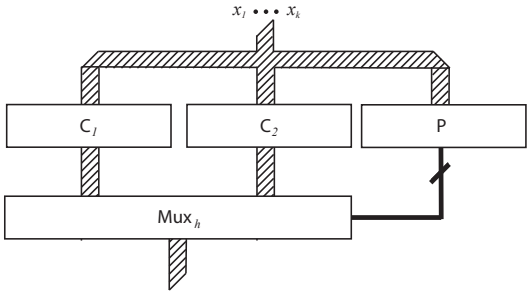


$x_{x_i} = \phi_m(x_{x_j}, x_{x_l})$ სქემა

აქ ცვლადების მიმდევრობა გამოსახულია, როგორც დაშტრიხული ზოლი, რომელსაც ჩვენ „მონაცემთა ზოლს" ვუოდებთ. ამ ზოლიდან შესაძლებელია ნებისმიერ ადგილზე საჭირო ინფორმაციის (ამ შემთხვევაში ცვლადების შესაბამისი მავთულების) გამოტანა. Select$_h$ ისეთი სქემაა, რომელიც ზემოდან მიღებული ცვლადებიდან (ამ შემთხვევაში ყველა ცვლადიდან) აირჩევს მეიმდენეს, რაც მარჯვენა სიგნალებშია კოდირებული, ხოლო Store$_n$ მისი შებრუნებულია: ზემოდან იღებს ყველა ცვლადს და ასევე გამოთვლის შედეგს, ქვემოთ კი გამოუშვებს უცვლელად ყველა ცვლადს ერთის გარდა: მარჯვნივ მიღებულ ინფორმაციაში კოდირებული რიცხვის პოზიციაზე მდგარ ცვლადს შეცვლის გამოთვლის შედეგით.
საბოლოო ჯამში მივიღებთ ცვლადების აქტუალურ მიმდევრობას.
თუ ალგორითმის ყოველ სტრიქონში ჩაწერილ ბრძანებას ამ მეთოდით გადავაკეთებთ სქემებად, მათი ერთი მეორეზე ვერტიკალურად გადაბმის შედეგად მივიღებთ ამ ალგორითმის შესაბამის სქემას, სადაც გამოსასვლელ სიგნალებში კოდირებული იქნება საწყისი ცვლადების საბოლოო მნიშვნელობები თავდაპირველად მოცემული მიმდევრობით.
თუ ალგორითმის ნაწილში ჩაწერილია

if(P)
then კოდი C_1
else კოდი C_2

მისი შესაბამისი სქემა იქნება



if(P) then C_1 else C_2 სქემა

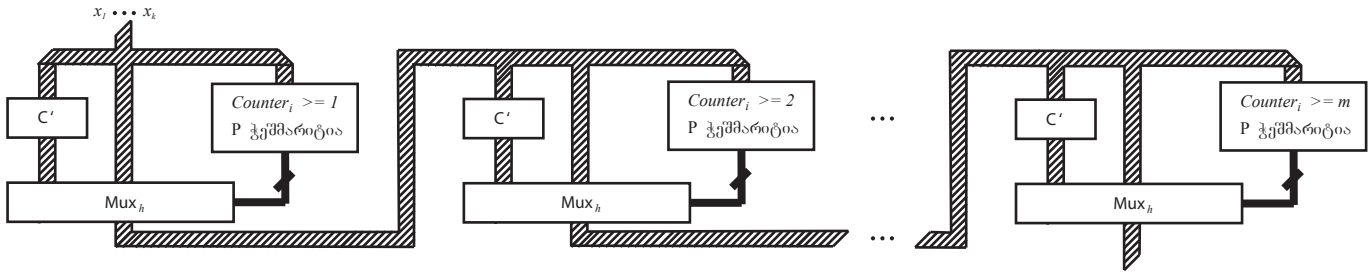
ბოლოს გასარკვევია, თუ როგორ შეიძლება ციკლის რეალიზაცია:

for(P)
{ კოდი C }

პირველ რიგში უნდა განვსაზღვროთ, თუ ***მაქსიმუმ*** რამდენჯერ შეიძლება დატრიალდეს ციკლი და C კოდის შესაბამისი სქემაც ამდენჯერ გადავაბათ ვერტიკალურად ერთმანეთს.
მეორე საკითხია, თუ ამ ჯაჭვის რომელი კომპონენტის გამოთვლა უნდა იყოს საბოლოო (ცხადია, რომ უმეტეს შემთხვევაში ციკლი უფრო ნაკლებჯერ დატრიალდება).
ამისათვის შემოვიღოთ ერთი დამატებითი ცვლადი $Counter_i$ (რომელიც ასევე საერთო ცვლადების მიმდევრობაში იქნება რომელიმე პოზიციაზე) და ციკლი შემდეგნაირად გადავწეროთ:

$Counter_i = 0$;
for(P)
{
$Counter_i = Counter_i + 1$;
კოდი C
}

ზემოთ ხსენებული ჯაჭვის m-ურ კომპონენტში მოწმდება პირობა $Counter_i \geq m$ და თუ ეს ასეა, პასუხად ამ კომპონენტის გამოთვლა იქნება გადაცემული, წინააღმდეგ შემთხვევაში კი – წინა კომპონენტის (ნახ. **??**)



for ციკლის სქემა

შენიშვნა: აქ ჩავთვალოთ, რომ $Counter_i = 0$ ოპერაცია უკვე რეალიზებულია და C' სქემა C კოდის რეალიზებასაც და მთვლელის გაზრდასაც მოიცავს. ამას გარდა, P გამონათქვამი ყველა შრისთვის სხვადასხვა ცვლადს მოიცავს და, აქედან გამომდინარე, რაღაც ადგილიდან დაწყებული მცდარი იქნება.

ცხადია, რომ ამ მეთოდით ნებისმიერი ალგორითმის სქემის შექმნა შეიძლება და ამ სქემის „შრეების" რაოდენობა ალგორითმის მაქსიმალური ბიჯების რაოდენობის მუდმივ ხარისხს (უფრო ზუსტად, კუბს) არ გადააჭარბებს.

რ.დ.გ.

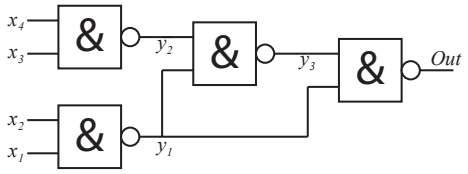
სავარჯიშო 6.6: ახსენით, თუ რის გამო იქნება ზემოთ მოყვანილი ჯაჭვის რაღაც ადგილამდე P გამონათქვამი ჭეშმარიტი და შემდეგ კი – ყველგან მცდარი. რა ტიპის პირობაა P?

სავარჯიშო 6.7: ახსენით, თუ რატომ შეიძლება გახდეს საჭირო ყოველ ციკლში თავისი მთვლელის შემოღება. რატომ არ შეიძლებოდა ერთი მთვლელით ოპერირება?

პირველი NP-სრული ამოცანის გამოყოფის შემდეგ უნდა აიგოს გარკვეული „ჯაჭვი" იმ ამოცანებთან, რომლებიც ჩვენ აქამდე განვიხილეთ და რომლებიც ერთი მეორეზე დავიყვანეთ. ამით დამტკიცდება ამ თითოეული ამოცანის NP-სრულობაც.
ლოგიკური იქნებოდა, თუ $CSAT$ ამოცანას SAT ამოცანაზე დავიყვანდით: მოცემული სქემისთვის ჩავწერდით შესაბამის ფუნქციას კონიუნქციური ნორმალური ფორმით. ცხადია, თუ ეს ფუნქცია დაკმაყოფილდება, მაშინ ასევე დაკმაყოფილდება თავდაპირველი სქემაც.
მაგრამ ეს მიდგომა გარკვეულ ხიფათს შეიცავს: როგორც წინა სემესტრის მასალიდანაა ცნობილი, ფუნქციის დიზიუნქციური ნორმალური ფორმის პოვნა ექსპონენციურ დროს მოითხოვს (ცვლადების რაოდენობასთან შედარებით), ჩვენ კი პოლინომური დაყვანა გვაინტერესებს.

ამ საკითხის მოგვარებაში გვეხმარება ე.წ. „ცაიტინის გარდაქმნა" (Tseytin transformation).
თეორემა 2.2: მოცემულია სქემა, რომელიც იყენებს მხოლოდ $NAND$ კომპონენტს (ანუ $\overline{x \cdot y}$). მისი შესაბამისი ფუნქცია პოლინომურ დროში ჩაიწერება კონიუნქციური ნორმალური ფორმით.

დამტკიცება: $z = \overline{x \cdot y}$ გამოსახულებაში შემავალი x, y, z სამეულის ურთიერთდამოკიდებულება (ანუ რა კომბინაციებია დასაშვები) გამოიხატება ფუნქციით $(x \vee z)(y \vee z)(\bar{x} \vee \bar{y} \vee \bar{z})$ (ეს გამოსახულება იღებს მნიშვნელობას 1 მაშინ და მხოლოდ მაშინ, თუ z ცვლადის მნიშვნელობა x და y ცვლადების მნიშვნელობათა კონიუნქციის უარყოფაა).
განვიხილოთ მოცემული C სქემა (მაგალითისთვის განვიხილოთ სქემა, რომელიც ნაჩვენებია ნახაზში **??**).



NAND ელემენტებზე აგებული სქემა ცვლადებით

ყოველი $NAND$ ელემენტის გამომავალი სიგნალიც ცალკე ცვლადით მოვნიშნოთ, ხოლო ამ სქემის გამომავალ სიგნალს ვუწოდოთ Out.
მაშინ ქვედა მარცხენა $NAND$ ელემენტის აღწერა შეიძლება გამოსახულებით $(x_1 \vee y_1)(x_2 \vee y_1)(\bar{x}_1 \vee \bar{x}_2 \vee \bar{y}_1)$, მარცხენა ზედასი – $(x_3 \vee y_2)(x_4 \vee y_2)(\bar{x}_3 \vee \bar{x}_4 \vee \bar{y}_2)$, შუა ელემენტის როგორც $(y_1 \vee y_3)(y_2 \vee y_3)(\bar{y}_1 \vee \bar{y}_2 \vee \bar{y}_3)$ და ბოლო ელემენტის – $(y_1 \vee Out)(y_3 \vee Out)(\bar{y}_1 \vee \bar{y}_3 \vee \overline{Out})$.
ქართული სიტყვებით რომ ვთქვათ, ყოველი გამოსახულება გვეუბნება: „y_1, y_2, y_3 და Out გამოთვლის კანონიერი შედეგია. იმისათვის, რომ გავარკვიოთ, იღებს თუ არა ეს სქემა ოდესმე მნიშვნელობას 1, საკმარისია ამ გამოსახულებების კონიუნქციებით შეკვრა და – რადგან გვაინტერესებს, არის თუ არა შედეგი 1 – თვით Out ცვლადიც კონიუნქციით უნდა მივუწეროთ. შედეგად ვიღებთ:

$$(x_1 \vee y_1)(x_2 \vee y_1)(\bar{x}_1 \vee \bar{x}_2 \vee \bar{y}_1) \wedge$$
$$(x_3 \vee y_2)(x_4 \vee y_2)(\bar{x}_3 \vee \bar{x}_4 \vee \bar{y}_2) \wedge$$
$$(y_1 \vee y_3)(y_2 \vee y_3)(\bar{y}_1 \vee \bar{y}_2 \vee \bar{y}_3) \wedge$$
$$(y_1 \vee Out)(y_3 \vee Out)(\bar{y}_1 \vee \bar{y}_3 \vee \overline{Out}) \wedge Out$$

სავარჯიშო 6.8: დაამტკიცეთ, რომ ამ მეთოდით შედგენილი კონიუნქციური ნორმალური ფორმა მოგვცემს საწყისი სქემის ფუნქციას.

სავარჯიშო 6.9: დაამტკიცეთ, რომ ნებისმიერი სქემა შეგვიძლია გადავიყვანოთ მხოლოდ NOR ელემენტებით აგებულ სქემაში (კონიუნქცია, დიზიუნქცია და უარყოფა აღწერეთ NOR ელემენტების გამოყენებით).

ამ სავარჯიშოებით მთელი დამტკიცება დასრულებულია.

რ.დ.გ.

თავი 7

არაამოხსნადობისა და NP სრული ამოცანების პრაქტიკული მნიშვნელობა

NPკლასისა და მისთვის რთული ან სრული ამოცანების უკეთ შესწავლისთვის საჭიროა ასევე იმის გააზრება, თუ რა კავშირშია იგი მის გარეთ მდგომ (კერძოდ კი არაამოხსნად ამოცანათა) სიმრავლესთან. აქ არსებობს ერთი მნიშვნელოვანი ფაქტი, რომელსაც ჩვენ ახლა განვიხილავთ.

7.1 შეჩერების ამოცანის NP-რთულობა

საინტერესოა ის ფაქტი, რომ არსებობს ისეთი არაამოხსნადი ამოცანა (კერძოდ კი შეჩერების ამოცანა H), რომელზედაც ყველა რეკურსიული ამოცანა დაიყვანება. ეს კი იმას ნიშნავს, რომ შეჩერების ამოცანა ყველა ამოხსნად ამოცანაზე „რთული" უნდა იყოს. რა თქმა უნდა, ეს ფაქტი პირდაპირ პრაქტიკულ შედეგს ვერ მოგვცემს, რადგან არაამოხსნადი ამოცანის რეალიზაციაც არაფერს მოგვცემს, მაგრამ მისი გამოყენებით მნიშვნელოვანი თეორიული და, აქედან გამომდინარე, პრაქტიკული შედეგების მიღება შეიძლება.
სანამ კონკრეტული თეორემის დამტკიცებას განვიხილავთ, ჯერ უნდა გავიაზროთ, თუ რას ნიშნავს ერთი ამოცანის მეორეზე დაყვანა ამ ამოცანების შესაბამისი ენის ტერმინებში.
განვიხილოთ სიმრავლე L_H, რომელიც ისეთი K ალგორითმისა და X მონაცემის (K, X) წყვილებისგან შედგება, რომ $K(X)$ აუცილებლად შეჩერდება.
ახლა კი განვიხილოთ ნებისმიერი რეკურსიული A ამოცანის შესაბამისი ენა L_A, რომელიც იმ X მონაცემებისგან შედგება, რომელთათვისაც $A(X) =$„კი".
დაყვანის ალგორითმით A ამოცანის X მონაცემი უნდა გადავაკეთოთ ისეთ A' ალგორითმსა და X' მონაცემში, რომ $X \in L_A \Leftrightarrow (A', X') \in L_A$.

ამის გათვალისწინებით დავამტკიცოთ

თეორემა 1.1: შეჩერების ამოცანა NP-რთულია.

დამტკიცება: განვიხილოთ ნებისმიერი გადაწყვეტილების ამოცანა NPკლასიდან. რადგან იგი რეკურსიულია, უნდა არსებობდეს შესაბამისი ალგორითმი A, რომელიც მას ამოხსნის (იმ ფაქტს, რომ ეს ალგორითმი ძალიან ნელი - ექსპონენციურიც კი შეიძლება იყოს, აქ არანაირი მნიშვნელობა არ აქვს, რადგან ჩვენ უბრალოდ ამოხსნადობა და არა მისი სისწრაფე გვაინტერესებს).
ჩვენ გვაინტერესებს, თუ რა პასუხს მოგვცემს A ალგორითმი X მონაცემზე, ანუ $A(X)$.
შევქმნათ ალგორითმი H_A შემდეგნაირად:

ალგორითმი 21: H_A
მოცემულია: ალგორითმი A და მისი მონაცემი X

```
1: w = A(X);
2: while(¬w)
```

ცხადია, რომ ეს ალგორითმი შეჩერდება მაშინ და მხოლოდ მაშინ, თუ A ალგორითმი X მონაცემზე მოგვცემს პასუხს „კი". აქედან გამომდინარე, თავდაპირველი ამოცანა დავიყვანეთ შეჩერების ამოცანაზე: თუ H_A შეჩერდება, თავდაპირველი ამოცანის პასუხი ყოფილა „კი" და თუ არ შეჩერდება, თავდაპირველი ამოცანის პასუხი იქნებოდა „არა".

ბუნებრივია შეკითხვა: არსებობს თუ არა ისეთი ამოცანა NP კლასის მიღმა, რომელიც არ იქნება NP-რთული? თუ ამ კლასის გარეთ ყველა ამოცანა უფრო „რთულია", ვიდრე ამ კლასში შემავალი?
ცნობილი თეორემა, რომელსაც დაწვრილებით შემდგომში განვიხილავთ, გვეუბნება, რომ ნებისმიერი სირთულის ამოცანაზე უფრო რთული ამოცანის შედგენაა შესაძლებელი (ანუ „მჯობნის მჯობნი არ დაილევა"). მაგრამ აქ გვაინტერესებს, არსებობს თუ არა უფრო „სუსტი " ამოცანა NP კლასს გარეთ?
თუ განვიხილავთ ზემოთ ნახსენებ co-NP კლასის სრულ ამოცანებს, co-NP ≠NPდაშვებით, ეს ამოცანა ვერ იქნება NP-რთული. მაგრამ ეს მხოლოდ მნიშვნელოვანი დაშვებით, რაც ჯერ-ჯერობით უცნობია.

7.2 NP-სრული ამოცანების მნიშვნელობა

მას შემდეგ, რაც 1972 წელს რიჩარდ კარპმა თავის ცნობილ სტატიაში *SAT* ამოცანიდან გამომდინარე დაყვანის პრინციპით 21 ამოცანის NPსრულობა დაამტკიცა, ამ საკითხის მიმართ ინტერესი არ შენელებულა. ამის ძირითადი მიზეზი არის ის, რომ ნებისმიერი NPსრული ამოცანის სწრაფად ამოხსნა სხვა ნებისმიერი სერტიფიცირებადი ამოცანის სწრაფად ამოხსნას ნიშნავს და იმ დროს ბევრმა მკვლევარმა სწორედ რომელიმე NPსრული ამოცანის სწრაფად ამოხსნაში დაინახა მთავარი პრობლემა.
სამწუხაროდ, უამრავი მცდელობის მიუხედავად, ვერც ერთი ასეთი ამოცანა სწრაფად ვერ გადაჭრილა, თუმცა ამის იმედი ბევრჯერ გაჩენილა და დღესაც - თითქმის ნახევარი საუკუნის შემდეგ – კიდევ არ მილეულა, თუმცა საგრძნობლად კი შემცირდა.
მეორე მიზეზი, თუ რატომაა NP-სრულობა ასეთი მნიშვნელოვანი, მისი „უნივერსალურიბაა": რადგან ყველა ამოცანა ამ ტიპის პრობლემებზე დაიყვანება, ისინი ამა თუ იმ ფორმით იჩენენ თავს სხვადასხვა, ერთი შეხედვით შორს მდგომ ამოცანებშიც. აქედან გამომდინარე, თუ გვეცოდინება „გადაკეთებული" NPსრული ამოცანის რაიმე სახით გადაჭრის გზები, ეს შეიძლება ბევრი სხვა ამოცანის გადაჭრაში დაგვეხმაროს.
რადგან ერთი NPსრული ამოცანის სწრაფი ამოხსნა ყველა დანარჩენის სწრაფ ამოხსნასაც გამოიწვევს, ეს ფაქტი კი ძალიან გასაკვირი იქნებოდა NPსრული ამოცანების ერთმანეთისაგან საკმაოდ განსხვავებული სტრუქტურისა და ერთი შეხედვით „მარტივი" ამოცანებთან შედარებით აშკარა სირთულეების გამო, დღეისათვის მკვლევართა აბსოლუტური უმრავლესობა მიიჩნევს, რომ ამ კლასის ამოცანების ზუსტ ამოხსნაზე დროის კარგვა არაა მიზანშეწონილი და მათი გადაჭრის ალტერნატიულ გზებს ეძებს (მაგალითად მიახლოებითი ალგორითმები, სადაც ოპტიმიზაციის ამოცანას რაიმე პატარა მისაღები ცდომილებით ამოვხსნით, ან ალბათური ალგორითმებით, რომლითაც გამოთვლილი პასუხი დიდი ალბათობით სწორია, მაგრამ შესაძლებელია არასწორიც იყოს, მაგრამ ამ ალგორითმის ბევრჯერ გაშვებისას ყველაზე ხშირად მიღებული პასუხი პრაქტიკულად სწორი იქნება, თუმცა შეცდომის ალბათობა მაინც იარსებებს და ა.შ.).
ყოველივე ზემოთ თქმულიდან გამომდინარე, უცნობი ამოცანისთვის პირველ რიგში გასარკვევია, არის თუ არა იგი NPსრული და თუ არის, ამოხსნის ალტერნატიული გზებია საძებნი.

იმისათვის, რომ რაიმე ახალი ამოცანის NPსრულობა დამტკიცდეს, საჭიროა რაიმე ცნობილი NPსრული ამოცანის მასზე დაყვანა. თეორიულად ნებისმიერი ცნობილი სრული ამოცანის დაყვანა უნდა იყოს შესაძლებელი, მაგრამ, როგორც პრაქტიკამ გვაჩვენა, რაიმე კონკრეტულ ამოცანაზე ზოგი საწყისი NPსრული ამოცანიდან დაყვანა გაცილებით უფრო მარტივია, ვიდრე სხვა საწყისი ამოცანიდან.
გამოცდილებიდან გამომდინარე, ადვილი დაყვანის საპოვნელად ოთხ საწყის NPსრულ ამოცანას განიხილავენ:

- VC_k (წიბოების გადაფარვა): გამოიყენება ისეთ ამოცანებზე დასაყვანად, რომლებიც გრაფებში ამორჩევას ეხება;
- HP (ჰამილტონის გზა): ამოცანებისთვის გრაფებზე, რომლებიც დამოკიდებულია გადალაგებებსა და მარშრუტიზაციაზე;
- IP (რიცვთა დაყოფა: მოცემულია ნატურალურ რიცხვთა სიმრავლე; შეიძლება თუ არა მისი დაყოფა ორ ქვესიმრავლედ ისე, რომ თითოეულ ქვესიმრავლეში შესულ რიცხვთა ჯამი ტოლი იყოს?): ისეთი ამოცანებისთვის, რომელთა სირთულე დამოკიდებულია დიდ რიცხვებზე;

- 3-*SAT* (სამცვლადიანი დნფ ფუნქციების შესრულებადობა): ისეთი ამოცანებისთვის, რომლებსაც არც ერთი ზემოთ ჩამოთვლილი საწყისი ამოცანა არ შეესაბამება.

სავარჯიშო 7.1: განიხილეთ ზურგჩანთის დისკრეტული ამოცანა: მოცემულია ზურგჩანთა W მოცულობით და n ცალი ტვირთი, თითოეული მოცულობით $a_1, ..., a_n$. შეიძლება თუ არა $a_{i_1}, ..., a_{i_k}$ ტვირთის შერჩევა ისე, რომ ზურგჩანთა მთლიანად შეივსოს (ანუ $a_{i_1} + \cdots + a_{i_k} = W$)?
დაამტკიცეთ ამ ამოცანის NPსრულობა.

სავარჯიშო 7.2: განიხილეთ ზურგჩანთის ამოცანის დინამიურ დაპროგრამებაზე დაფუძნებული ალგორითმი (იხ. წინა სემესტრის მასალა).
რა არის ამ ალგორითმის დროის ზედა ზღვარი?

სავარჯიშო 7.3: განიხილეთ ქვეგრაფის იზომორფულობის ამოცანა *SGI*. რომელი საწყისი ამოცანიდანაა მისი NPსრულობის დამტკიცება ყველაზე მოსახერხებელი? დაიყვანეთ ეს ამოცანა ქვეგრაფის იზომორფიზმზე.

სავარჯიშო 7.4: დაამტკიცეთ შემდეგი ამოცანის NPსრულობა.
პროგრამათა ექვივალენტურობა: მოცემულია ცვლადების სასრული სიმრავლე X და მნიშვნელობათა სასრული სიმრავლე V, ასევე ორი პროგრამა P_1 და P_2, რომელიც შემდეგი ტიპის ბრძანებებისგან შედგება:

$x_0 \longleftarrow \text{if}(x_1 = x_2) \text{ then } x_3 \text{ else } x_4$

არსებობს თუ არა X სიმრავლის ცვლადების ისეთი მნიშვნელობები V სიმრავლიდან, რომ P_1 და P_2 პროგრამა სხვადასხვა პასუხს იძლეოდეს? ან, სხვა სიტყვებით რომ ვთქვათ, იძლევა თუ არა ეს ორი პროგრამა ერთსა და იმავე მონაცემზე ერთსა და იმავე პასუხს?

7.3 არაგამოთვლადი ფუნქციების გამოყენება

რაც არ უნდა გასაკვირი იყოს, არაგამოთვლადი ფუნქცია შეიძლება ძალიან მნიშვნელოვან როლს თამაშობდეს პრაქტიკაში. ერთ-ერთი ასეთი გამოყენების სადემონსტრაციოდ განვიხილოთ ე.წ. „საქმიანი თახვის" ამოცანა.
როგორც ცნობილია, თახვები უაღრესად ბეჯითი ცხოველები არიან: გადადიან ტყეში, იღებენ მორებს, მდინარეში მიცურავენ და წყალსაგუბებლებს აშენებენ. ასეთ იქით-აქეთ სირბილს თახვი თავს არ ანებებს, სანამ თავის სამუშაოს ბოლომდე არ დაასრულებს. თახვების მსგავსად მოქმედებს ალგორითმიც: გამოთვლის პროცესში ცვლადებს შორის იქით-აქეთ დარბის და მეხსიერებაში გარკვეულ სიმბოლოებს წერს.
1962 წელს უნგრელმა მათემატიკოსმა ტიბორ რადომ წამოაყენა „საქმიანი თახვის" ამოცანა: მოცემულია ორობით კოდში ჩაწერილი n სიგრძის ალგორითმი, რომელიც მუშაობას ცარიელი სიტყვით (როგორც მონაცემი) და ნულის ტოლი ცვლადებით იწყებს და როდესღაც ჩერდება. რამდენი 1 იქნება დაწერილი მეხსიერებიაში მანქანის გაჩერების შემდეგ? და ზოგადად: მაქსიმუმ რამდენი 1 შეიძლება დაწეროს n სიგრძის ალგორითმმა გაჩერებამდე, თუ იგი მუშაობას ცარიელი მეხსიერებით (ნულის ტოლი ცვლადებით) დაიწყებს? ამ რიცხვს გამოსახავენ $\Sigma(n)$ ფუნქციით, რომელსაც ***რადოს ფუნქციას*** უწოდებენ. ანალოგიურად შეიძლება დავსვათ შეკითხვა: მაქსიმუმ რამდენი ბიჯის შემდეგ გაჩერდება n სიგრძის ალგორითმი? ეს რიცხვი აღინიშნება ფუნქციით $S(n)$ (თუ ალგორითმი არ შეჩერდება, მაშინ $S(n) = 0$). როგორც დამტკიცებულია, ეს ორივე ფუნქცია უფრო სწრაფად იზრდება, ვიდრე *ნებისმიერი* გამოთვლადი ფუნქცია, ანუ ორივე არაგამოთვლადია:

n	1	2	3	4	5	6
$\Sigma(n)$	1	4	6	13	≥ 4098	$\geq 10^{1439}$
$S(n)$	1	6	21	107	$\geq 47.176.870$	$\geq 10^{2879}$

შენიშვნა: რადოს ფუნქცია განისაზღვრება ე.წ. ტიურინგის მანქანებით - გამოთვლის მოდელით, რომელიც სტანდარტულადაა მიჩნეული. აქ ჩვენ მის ანალოგს განვიხილავთ, რომელიც ინტუიციურ დონეზე (სიმკაცრის ხარჯზე) შედეგების მნიშვნელობას წარმოაჩენს.

ამ ფუნქციების გამოუთვლადობის მიუხედავად, უაღრესად დიდი მნიშვნელობა აქვს მათ დათვლას კონკრეტული n-თვის. მაგალითად, თუ შევადგენთ ზემოთ ნახსენებ ალგორითმს, რომელიც რიგ-რიგობით გადაამოწმებს ნატურალურ რიცხვებს და გაჩერდება მაშინ და მხოლოდ მაშინ, თუ ისეთ ლუწ რიცხვს აღმოაჩენს, რომელიც არ წარმოდგება ორი მარტივი რიცხვის ჯამის სახით, მაშინ შეიძლება გოლდბახის ჰიპოთეზის შემოწმება: დავითვლით

ამ ალგორითმის სიგრძეს n, გამოვითვლით $S(n)$ ფუნქციას და ამ მანქანას ვამუშავებთ $S(n)$ ბიჯის განმავლობაში. თუ იგი ამ დროში არ შეჩერდა, ესე იგი აღარასდროს შეჩერდება და ჩვენ რიცხვთა თეორიის ერთ-ერთ უმნიშვნელოვანეს საკითხს გადავჭრით.
სამწუხაროდ, ჯერ-ჯერობით ასეთი პერსპექტივა საკმაოდ ბუნდოვანია.

სავარჯიშო 7.5: დაამტკიცეთ, რომ $S(n)$ ფუნქცია არაა გამოთვლადი.
მინიშნება: შეჩერების ამოცანა დაიყვანეთ $S(n)$ ფუნქციის გამოთვლის ამოცანაზე.

სავარჯიშო 7.6: დაამტკიცეთ შემდეგი გამონათქვამის ჭეშმარიტება:
(A არაგამოთვლადია, $A \leq_p B$) $\Leftrightarrow$ (B არაგამოთვლადია)

ამ სავარჯიშოდან გამომდინარეობს მნიშვნელოვანი თეორემა:

თეორემა 3.2: ნებისმიერი არაგამოთვლადი ამოცანა ნებისმიერ გამოთვლად ამოცანაზე რთულია.

თავი 8

P vs. NP

თანამედროვე მეცნიერების ერთ-ერთი უმნიშვნელოვანესი ღია საკითხია P vs. NP ამოცანა, რომლის ჩამოყალიბებასაც, ფაქტიურად, წინა თავების მასალა მივუძღვენით. იმ ფაქტიდან გამომდინარე, რომ მკაცრად ვერც ტოლობისა და ვერც უტოლობის დამტკიცება ვერ მოხერხდა, მეცნიერებმა იმაზე დაიწყეს ფიქრი, თუ რა მოხდება ამა თუ იმ ვარიანტის ჭეშმარიტების შემთხვევაში. ამ საკითხების გამოკვლევას ეძღვნება ამ თავის თემა, სადაც ბოლოში ერთი ძალიან მნიშვნელოვანი თეორემა დამტკიცდება შუალედური ენის არსებობის შესახებ.

8.1 P vs. NP ამოცანა და მისი გადაჭრის ვარიანტები

P vs. NP ამოცანის პასუხის სხვადასხვა ვარიანტი არსებობს:

1. P=NP: ამ შემთხვევაში ყველა ამოცანა, რომელიც პოლინუმურ დროში სეტრიფიცირებადია, ასევე პოლინომურ დროში ამოხსნადი იქნება. მაგრამ ეს სულაც არ ნიშნავს იმას, რომ აღვილი იქნება ასეთი ალგორითმების პოვნა. ეს პირველ რიგში ე.წ. „არსებობის თეორემა" იქნება, ანუ გვეცოდინება, რომ სწრაფი ამოხსნა შესაძლებელია, მაგრამ ასეთი ამოხსნის პოვნა ცალკე ამოცანაა. რა თქმა უნდა, შესაძლებელია ე.წ. „კონსტრუქციული დამტკიცება", ანუ ვინმე რომელიმე NPსრული ამოცანისთვის დაწერს სწრაფ ალგორითმს, რითიც ტოლობა დამტკიცდება და ასევე კონკრეტული პოლინომური ამოხსნაც იქნება მოცემული, თუმცა ასეთი პერსპექტივა დღევანდელი მიღწევებიდან გამომდინარე საკმაოდ შორია.

   ხუმრობით იმასაც ამბობენ, რომ P = NPშემთხვევაში ვინც გაუსის დამტკიცებას გაიგებს, ის გაუსივით ჭკვიანი იქნება.

2. P ≠ NPეს შედეგი იმის მაჩვენებელი იქნება, რომ არსებობს ისეთი (არაერთი) სწრაფად სერტიფიცირებადი ამოცანა, რომლის სწრაფად ამოხსნა შეუძლებელია. პრაქტიკაში ეს იმას უნდა ნიშნავდეს, რომ NP- სრული ამოცანებისთვის ამოხსნის თავისებური გზებია საძებნი: ან მიახლოებითი ამოხსნა (ოპტიმიზაციის ამოცანებში), ან მონაცემთა შეზღუდვა, ან ე.წ. ევრისტიკის ძიება და სხვა. ამ საკითხებს ცოტა მოგვიანებით შევეხებით.

3. ორივე დაშვება *თავის თავად და ერთმანეთისგან დამოუკიდებლად,* ჭეშმარიტია. ასეთი შემთხვევები მათემატიკაში ცნობილია: ნებისმიერი მათემატიკური თეორია დაფუძნებულია აქსიომებზე, რომელთა ლოგიკური კომბინაციებით მიიღება თეორემები, მათი ლოგიკური კომბინაციით კიდევ დამატებითი თეორემები და ამგვარად შეიძლება თეორიის გაფართოება. მთავარი ის არის, რომ ლოგიკური დასკვნის ჯაჭვით არ მივიღოთ ორი ურთიერთგამომრიცხავი თეორემა (ეს არის ლოგიკის ერთ-ერთი ძირითადი პრინციპი: ერთ თეორიაში არ უნდა მტკიცდებოდეს რაიმე გამონათქვამი და მისი უარყოფა. ამ შემთხვევაში იტყვიან, რომ ეს სისტემა არაწინააღმდეგობრივია). თუ ვინმე შეძლებს გამოთვლის თეორიის აქსიომების შექმნას, საიდანაც (ცნობილი) თეორემების გამოყვანა შეიძლება, ამ აქსიომებს P ≠ NP თეზისს მიუმატებს და დაამტკიცებს, რომ მიღებული სისტემა არაწინააღმდეგობრივია, მაშინ შეიძლება იმის დაჯერება, რომ P = NPჭეშმარიტია. მაგრამ, ამის და მიუხედავად, თუ იგივე აქსიომებს დავუმატებთ P ≠ NPაქსიომას და ასევე არაწინააღმდეგობრივ სისტემას მივიღებთ, ასევე იარსებებს თეორია, რომელშიც P ≠ NP. ეს ორი თეორია *ერთმანეთისგან დამოუკიდებლად* ჭეშმარიტი იქნება.

ეს საკმაოდ რთული საკითხებია, რომელთა დაწვრილებით შესწავლა ჩვენი კურსის ფარგლებს ცდება, თუმცა სპეციალური კურსების ფარგლებში განიხილება.

8.2 P ≠ NP დაშვებიდან გამომდინარე შედეგები

NP-სრული ამოცანებისთვის სწრაფი ამოხსნის თითქმის ორმოცდაათი წლის განმავლობაში წარუმატებელი ძიების შედეგად ალგორითმების თეორიითა და პრაქტიკით დაინტერესებული პირები იძულებილები გახდნენ, თავიანთი კვლევა P ≠ NP დაშვების შედეგად ეწარმოებინათ: დავუშვათ, რომ P ≠ NP. როგორ ამოვხსნათ რთული ამოცანები? რა სტრუქტურა ექნება NPდა co-NP სიმრავლეებს? რა დამოკიდებულებაა მათ შორის? ამ საკითხებზე შემდგომში გვექნება ლაპარაკი.

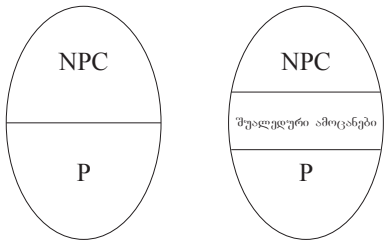
აღსანიშნავია ის ფაქტიც, რომ თუ წინა პარაგრაფში განხილული შესაძლებლობების მესამე პუნქტს განვიხილავთ, ანუ ორივე შესაძლებლობა იქნება დასაშვები და P = NP შემთხვევას მივიღებთ, NP-სრული ამოცანებისთვის მაინც ვერ შევძლებთ კონკრეტული სწრაფი ალგორითმის დაწერას.

<u>სავარჯიშო 8.1:</u> დაამტკიცეთ, რომ თუ არც P = NPდა არც P ≠ NPგამოიწვევს წინააღმდეგობას და ამ შემთხვევაში დავუშვებთ, რომ P = NP, NP-სრული ამოცანებისთვის სწრაფი ალგორითმის არსებობა ჭეშმარიტი იქნება, მაგრამ ასეთ ალგორითმს ვერავინ აღმოაჩენს.

ასევე საინტერესოა, რომ P ≠ NP დაშვების პირობებში *შესაძლებელია,* ამოცანა „დაამტკიცეთ P ≠ NP" თვითონ NP-სრული აღმოჩნდეს და, აქედან გამომდინარე, დამტკიცების პოვნაც რეალურ დროში შეუძლებელი იყოს.

8.2.1 შუალედური ენის არსებობა

P ≠ NP დაშვების პირობებში NPსიმრავლის ორი ვარიანტი არსებობს, რომელიც ნაჩვენებია სურათში **??**.



(ა) (ბ)

NP სიმრავლის შესაძლო სტრუქტურა P ≠ NP დაშვების პირობებში
ან NP სიმრავლე ზუსტად ორ P და NPC კლასად იყოფა, ან უნდა არსებობდეს ისეთი ენა (ამოცანა), რომელიც არ ამოიხსნება პოლინომურ დროში და არც NP-სრულია.
შემდეგი თეორემა სწორედ ამ ფაქტს ამტკიცებს.

<u>თეორემა 2.1:</u> თუ P ≠ NP, მაშინ არსებობს ისეთი ენა NP სიმრავლეში, რომელიც არ ეკუთვნის P სიმრავლეს და <u>*ამავდროულად*</u> არ არის NP-სრული.

<u>დამტკიცება:</u> რადგან ალგორითმების სიმრავლე თვლადია, შესაძლებელია ყველა ისეთი $M_1, M_2, ...$ ალგორითმის გადანომვრა, რომელთა მუშაობის დრო ზემოდან პოლინომური ფუნქციითაა შემოსაზღვრული. ასევე შეიძლება $R_1, R_2, ...$ პოლინომურ დროში დაყვანათა ალგორითმების გადანომვრაც. რადგან ეს შესაძლებელია, უნდა არსებობდეს ალგორითმი TM_1, რომელიც რიგ-რიგობით მოგვცემს ასეთ M_i მიმდევრობას; ანალოგიურად უნდა არსებობდეს TM_2 ტიურინგის მანქანაც, რომელიც რიგ-რიგობით მოგვცემს R_i მიმდევრობას (ანუ M_i და, შესაბამისად, R_i ალგორითმების აღწერას).

შენიშვნა: აღსანიშნავია, რომ ამდაგვარი გადათვლის პოვნა ენათა ყველა კლასისთვის შესაძლებელი არაა. არსებობს ისეთ ენათა კლასიც, რომელთათვისაც გადათვლის პოვნა შესაძლებელი, მაგრამ ძალიან ძნელია.

ამას გარდა, TS იყოს ისეთი ალგორითმი, რომელიც გადაჭრის SAT ამოცანას (რადგან დავუშვით, რომ P ≠ NP, ეს ალგორითმი პასუხს ექსპონენციურ დროში გამოიანგარიშებს).
ახლა კი აღვწეროთ ისეთი L ენა, რომელიც არ იქნება P სიმრავლეში და არც NP- სრულია. ასეთ ენას ჩვენ პირდაპირ ვერ აღვწერთ, ამიტომ მოვიყვანთ ისეთი TK ალგორითმის მაგალითს, რომელიც მას გადაწყვეტს (მხოლოდ მის სიტყვებს მიიღებს), რითაც დავამტკიცებთ, რომ ასეთი ენა უნდა არსებობდეს.

თუ მოცემულია სპეციალური ფუნქცია $f : \mathbb{N} \to \mathbb{N}$ (რომლის განსაზღვრაც თეორემის დამტკიცების მთავარი ნაწილია), ზემოთ აღნიშნული ალგორითმის ფუნქცია (რომელსაც ჩვენ ასევე აღვნიშნავთ, როგორც TK) შემდეგნაირად განისაზღვრება:

$$TK(x) = \begin{cases} \text{„კი”} \\ \text{„არა”} \end{cases}, \text{ თუ } TS(x) = \text{ „კი” და ამავდროულად } f(|x|) \text{ ლუწია}$$

სხვა სიტყვებით რომ ვთქვათ, TK მხოლოდ მაშინ იღებს x სიტყვას, თუ x მოცემულია კონიუნქციური ნორმალური ფორმით, ცვლადების რომელიმე კომბინაციაზე იგი 1 ხდება (სრულდება) და მის სიგრძეზე გამოთვლილი f ფუნქცია ლუწია.

<u>სავარჯიშო 8.2:</u> ფორმალურად განსაზღვრეთ TK ალგორითმის ენა L_{TK}.

რაც შეეხება f ფუნქციას, იგი მონოტონურად ზრდადია, ანუ $f(n+1) \geq f(n)$ და $f(0) = 0$. თანაც f ძალიან ნელა იზრდება.
განვსაზღვროთ ალგორითმი TF, რომელიც ამ ფუნქციას გამოითვლის. TF ორ ეტაპად მუშაობს, თანაც ყოველი ეტაპი n ბიჯს გრძელდება.
პირველ ფაზაში TF ცდილობს მაქსიმალურად ბევრი $f(0), f(1), f(2), \ldots$ გამოანგარიშებას (რამდენსაც მოასწრებს n ბიჯში). დავუშვათ, რომ ბოლოს გამოთვლილი სიდიდე იყო $f(i) = k$. $f(n)$ სიდიდე იქნება ან k, ან $k+1$ იმის მიხედვით, თუ როგორ განვითარდება გამოთვლა მეორე ფაზაში.
მეორე ფაზაში მუშაობა იმაზეა დამოკიდებული, პირველ ფაზაში გამოთვლილი k რიცხვი კენტია თუ ლუწი. დავუშვათ, რომ $k = 2i$ ლუწია. მაშინ TF ითვლის $M_i(z)$, $S(z)$ და $TF(|z|)$, სადაც z რიგ-რიგობით გაირბენს ლექსიკოგრაფიულად დალაგებული Σ^* სიმრავლის ყველა ელემენტს და შეჩერდება მაშინ და მხოლოდ მაშინ, თუ აღმოაჩენს ისეთ z სიტყვას, რომ

$$TK(z) \neq M_i(z).$$

TK მანქანის განსაზღვრების თანახმად, იძებნება ისეთი z სიტყვა, რომლისთვისაც

- $M_i(z) =$„კი” და $S(z) =$„არა” ან $f(|z|)$ კენტია;
- $M_i(z) =$„არა” და $S(z) =$„კი” და $f(|z|)$ ლუწია.

აქაც გამოთვლა n ბიჯის შემდეგ წყდება და თუ ასეთი z არ აღმოჩნდა, $f(n) = k$. თუ ამ n ბიჯში ასეთი z აღმოჩნდა, მაშინ $f(n) = k+1$.

თუ პირველ ფაზაში გამოთვლილი $k = 2i+1$ კენტია, მაშინ მეორე ფაზა შემდეგნაირად ვითარდება: TF ითვლის $R_i(z)$, $S(R_i(z))$ და $F(|T_i(z))|)$, სადაც z ასევე ლექსიკოგრაფიულად გარბის Σ^* სიმრავლის ყველა მნიშვნელობას. აქ უკვე TF ეძებს ისეთ z სიტყვას, რომლისთვისაც

$$TK(R_i(z)) \neq S(z).$$

უნდა აღინიშნოს, რომ R_i დაყვანაა და, აქედან გამომდინარე, სიტყვას იძლევა.

TK მანქანის განსაზღვრების თანახმად, იძებნება ისეთი z სიტყვა, რომლისთვისაც

- $S(z) =$„კი” და ან $S(R_i(z)) =$„არა” ან $f(|z|)$ კენტია;
- $S(z) =$„არა” და $S(R_i(z)) =$„კი” და $f(|z|)$ ლუწია.

თუ გამოთვლისთვის გამოყოფილ n ბიჯში ასეთი z მოიძებნა, მაშინ $f(n) = k+1$. წინააღმდეგ შემთხვევაში $f(n) = k$.

აღსანიშნავია, რომ TF მკაცრად განსაზღვრული მანქანაა და $O(n)$ ბიჯში გამოითვლის $f(n)$ ფუნქციას.
აქედან გამომდინარე, ასევე TK იქნება მკაცრად განსაზღვრული.
<u>სავარჯიშო 8.3:</u> აჩვენეთ, რომ $L_{TK} \in \text{NP}$.

ახლა დავუშვათ, რომ $L_{TK} \in \text{P}$. მაშინ იარსებებს ისეთი M_i დასაწყისში ნახსენები გადანომვრის მიხედვით, რომლისთვისაც $TK(z) = M_i(z), \forall z$ (სხვა სიტყვებით რომ ვთქვათ, TK მანქანა ამ სიაში შეგვხვდება). მაგრამ ასეთ შემთხვევაში TF მანქანის მეორე ფაზაში არ აღმოჩნდება ისეთი z, რომლისთვისაც $TK(z) \neq M_i(z)$ და $f(n)$ ფუნქცია აღარ გაიზრდება და $f(n) = 2i, \forall n > n_0$. ეს კი იმას ნიშნავს, რომ დაწყებული რაღაც ადგილიდან L ენა

„გადაიზრდება” *SAT* ამოცანაში (ანუ $TK(x) = SAT(x)$) და, აქედან გამომდინარე, იქნება NPსრული, რაც მას P $\neq$ NPდაშვებით P სიმრავლიდან გამორიცხავს.

ახლა დავუშვათ, რომ $L \in$ NPC. მაშინ უნდა არსებობდეს დაყვანა R_i SAT ამოცანიდან L_{TK} ენაზე, რაც იმას ნიშნავს, რომ $TK(R_i(z)) = S(z), \forall z$. აქედან გამომდინარე, TF მანქანის მეორე ფაზაში $k = 2i+1$ შემთხვევისათვის შესაბამისი z არ მოიძებნება, რის შედეგადაც დაწყებული რაღაც ადგილიდან $f(n)$ არ გაიზრდება და დარჩება კენტი. აქედან გამომდინარე, L_{TK} ენა სასრული გამოვა, ანუ $L_{TK} \notin$ NPC.

ორივე დაშვებას ($L_{TK} \in$ P და $L_{TK} \in$ NPC) წინააღმდეგობამდე მივყავართ. აქედან გამომდინარე, L_{TK} „საშუალედო” უნდა იყოს, ანუ $L_{TK} \notin$ P და $L \notin$ NPC.

რ.დ.გ.

სავარჯიშო 8.4: დაამტკიცეთ, რომ ნებისმიერი სასრული ენა P სიმრავლეს ეკუთვნის.

სავარჯიშო 8.5: ზედა თეორემის დამტკიცების რომელ ადგილებში გამოვიყენეთ $P \neq$ NP?

P $\neq$ NP შემთხვევაში შუალედური ენების არსებობა ძალიან მნიშვნელოვანია პრაქტიკაშიც. მაგალითად, ნატურალურ რიცხვთა ფაქტორიზაციის ან გრაფთა იზომორფიზმის ამოცნისთვის ჯერ-ჯერობით ვერავინ მოიფიქრა პოლინომური ალგორითმი და ვერც მათი NP სრულობის დამტკიცება შეძლო. ამან გააჩინა საფუძვლიანი ეჭვი, რომ ეს ორი ამოცანა სწორედ ასეთ საშუალედო კლასს უნდა ეკუთვნოდეს, მაგრამ ეს მხოლოდ იმ პირობით, რომ P $\neq$ NP.

თავი 9

დინამიკური პროგრამირება

9.1 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.

9.2 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.

9.3 უგრძესი გზის პოვნა რიცხვების სამკუთხა ცხრილში

განვიხილოთ სურ. 9.1-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.

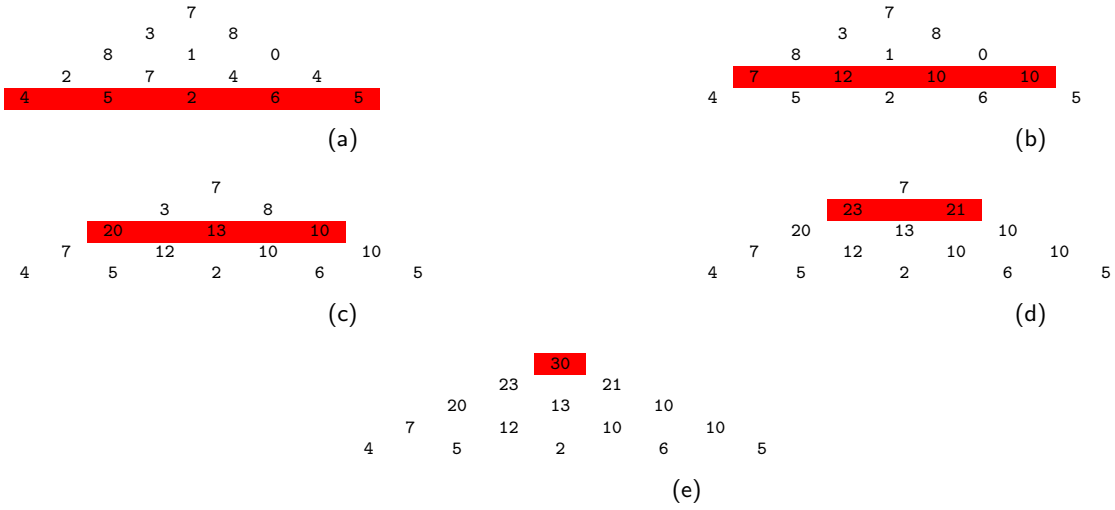
```
            7
          3   8
        8   1   0
      2   7   4   4
    4   5   2   6   5
```

ნახ. 9.1

- 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

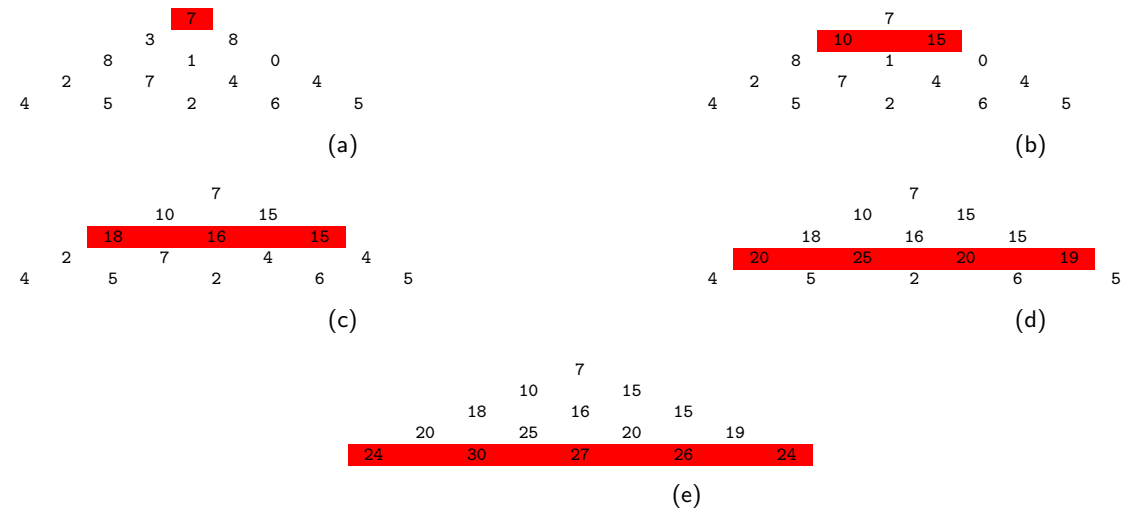
დიდი მოცულობის შემავალი მონაცემებისათვის მათი რეალიზაცია შესაძლებელია ერთგანზომილებიან მასივშიც, მაგრამ ამოცანაში მოცემული პარამეტრებისათვის ორგანზომილებიანი მასივის (ზომით 1000×1000) გამოყენებაც შეიძლება. მართალია, ასეთ მასივში ელემენტების დიდი ნაწილი NIL მნიშვნელობის იქნება (ჩვენი ამოცანისთვის გამოდგებოდა, 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. 9.2

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. 9.3-ზე მოცემულია ამ ალგორითმის მუშაობა (საწყისი სამკუთხედი იგივეა):



ნახ. 9.3

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 წევრიდან მოვედით მოცემულ ელემენტზე.
ანალოგიური ალგორითმით - ყოველ ბიჯზე მინიმალური ელემენტების შეკრებით, შესაძლებელია მინიმალური ჯამის პოვნაც.

9.4 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 9.1. (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. მტკიცდება 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 "წარმოშობის". $b[i, j]$ 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 22: 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. 9.4

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 23: 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. მაშასადმე, უსქ-ის პოვნა იგივე O(m+n) დროში ერთი ცხრილითაც შეიძლებოდა.

9.5 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? 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

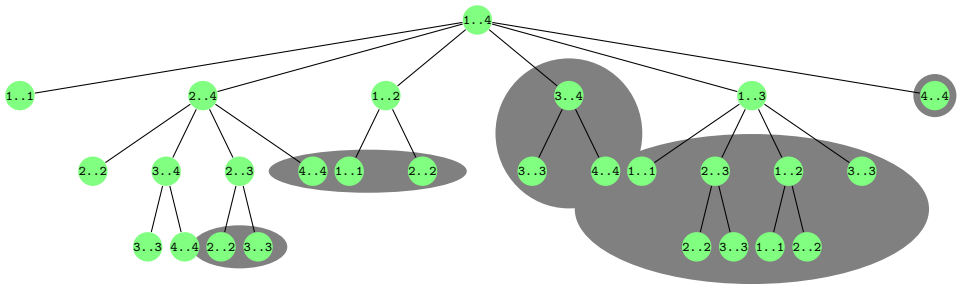
ამოხსნა რეკურსიული ხის სხვადასხვა განშტოებაში, სწორედ ამითაა განპირობებული მისი მუშაობის ექსპონენციალური დრო.

Algorithm 24: 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 25: 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. 9.5

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 26: 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

```
MEMOIZED-MATRIX-CHAIN(p) :
    n = len(p)-1;
    for i=1; i<=n; i++ :
        for j=i; j<=n; j++ :
            m[i][j] = ∞;
    return LOOKUP-CHAIN(p, 1, n);
LOOKUP-CHAIN(p, i, j) :
    if m[i][j] < ∞ :
        return m[i][j];
    if i == j :
        m[i][j] = 0;
    else:
        for k=i; k<=j-1; k++ :
            q = LOOKUP-CHAIN(p,i,k) + LOOKUP-CHAIN(p,k+1,j) + p[i-1]p[k]p[j];
            if q < m[i][j] :
                m[i][j] = q;
                s[i][j] = k;
    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 27: Print Optimal Parens

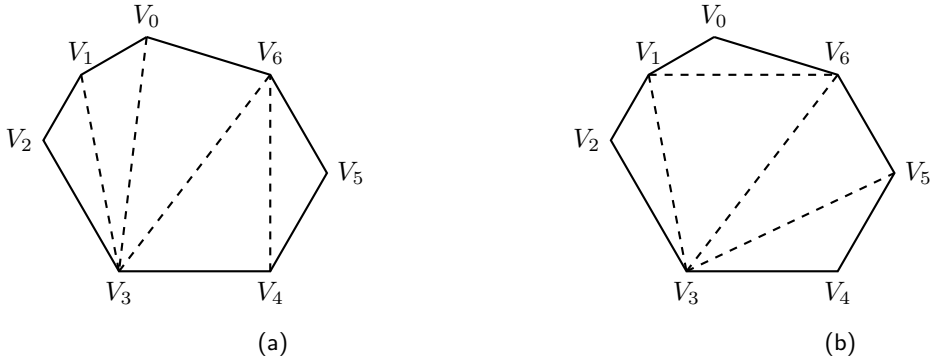
Input: MATRIX-CHAIN-ORDER(p)-თი გამოთვლილი s მატრიცა
Output: ბეჭდავს i-დან j-მდე მატრიცათა მიმდევრობის ოპტიმალური გადამრავლებისათვის საჭირო ფრჩხილების განლაგებას

```
PRINT-OPTIMAL-PARENS(s, i, j) :
    if i == j :
        print('A_i');
    else:
        print('(');
        PRINT-OPTIMAL-PARENS(s, i, s[i][j]);
        PRINT-OPTIMAL-PARENS(s, s[i][j]+1, j);
        print(')');
```

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

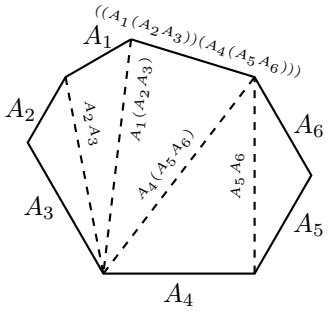
9.6 მრავალკუთხედის ოპტიმალური ტრიანგულაცია

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. 9.6

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. 9.7

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 $1 \le i \le j \le n$. 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 $j - i \ge 1$, მაშინ $< v_{i-1}, v_i, \ldots, v_j >$ მრავალკუთხედში გვაქვს არანაკლებ სამი წვეროსი და ყველა k-სათვის $i \le k \le j-1$ შეიძლება ვიპოვოთ ასეთი k 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:

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

შემდეგი მოქმედებები ანალოგიურია წინა ამოცანის მოქმედებებისა.

9.7 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 10

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.

10.1 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 28: 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;
```

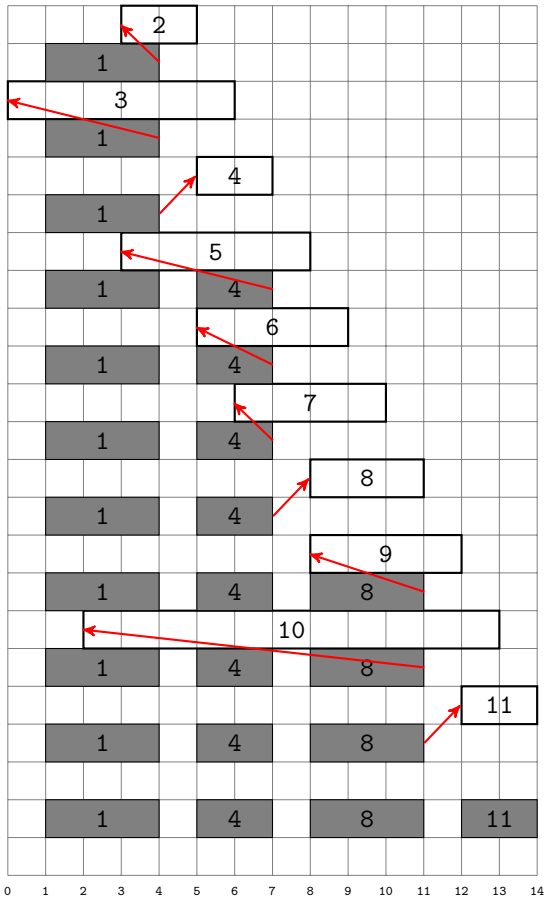
ალგორითმის მუშაობა მოცემულია მომდევნო ნახაზზე. A სიმრავლე შედგება ამორჩეული განაცხადების ნომრებისაგან, ხოლო j უკანასკნელია ამ ნომრებს შორის, ამასთან:

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

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

(a)



(b)

ნახ. 10.1:

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

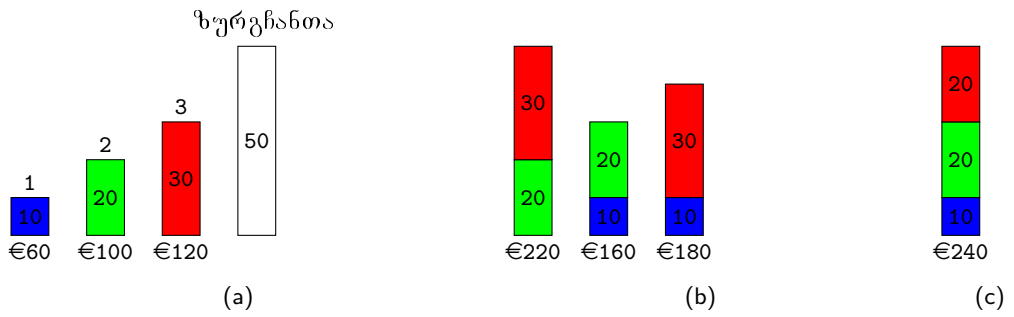
თეორემა 10.1. ალგორითმი GREEDY-ACTIVITY-SELECTOR გვაძლევს თავსებადი განაცხადების მაქსიმალურად შესაძლებელ რაოდენობას.

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. □

10.2 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 წყვეტი 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. 10.2:

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.

10.3 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).

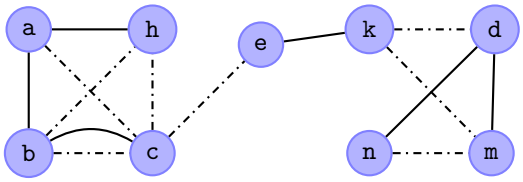
განვიხილოთ ორი მაგალითი.
ვთქვათ, S არის რომელიღაც მატრიცის სტრიქონების სიმრავლე, ხოლო რამდენიმე სტრიქონისგან შედგენილ სიმრავლეს ვუწოდოთ დამოუკიდებელი, თუ ეს სტრიქონები წრფივად დამოუკიდებელია ჩვეულებრივი აზრით. ადვილი საჩვენებელია, რომ ასე მიიღება მატროიდი. ეს მაგალითი იმითაა საინტერესო, რომ მატროიდების 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. 10.3:

ლემა 10.1. 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 10.2. 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 10.3. 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.

10.4 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 29: 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 დაურღვეველად, 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 10.2. (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 10.3. (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. □

10.4.1 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 10.4. 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 10.4. ვთქვათ, S არის ტოლი ხანგრძლივობის შეკვეთების სიმრავლე მოცემული შესრულების ვადებით, ხოლო I არის შეკვეთების დამოუკიდებელი ქვესიმრავლეების 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. 10.4:

10.5 სავარჯიშოები

თავი 11

ქვესტრიქონების ძებნის ამოცანა

11.1 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 ავღნიშნოთ —x—-ით. x და y სტრიქონების **კონკატენაცია** (concatenation), ავღნიშნოთ 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 11.1. (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 $T_n = T = T[1..n]$.

11.2 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).

11.3 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 30: 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 11.1: სულ 9 შედარება

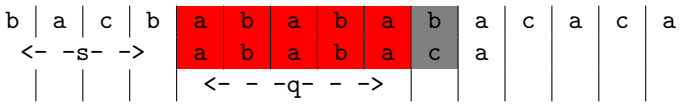
B	E	S	S	-	K	N	E	W	-	A	B	O	U	T	-	B	A	O	B	A	B	S
B	A	O	B	A	B																	
	B	A	O	B	A	B																
		B	A	O	B	A	B															
			B	A	O	B	A	B														
				B	A	O	B	A	B													
					B	A	O	B	A	B												
						B	A	O	B	A	B											
							B	A	O	B	A	B										
								B	A	O	B	A	B									
									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 11.2: 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.

11.4 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. π[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. π[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=bacbababababacaca.
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 11.3

ტაბულა 11.4

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.

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'ის, რომლისთვისაც:

$$P[1..k] = T[s' + 1..s' + k] \tag{11.1}$$

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 (11.1) ფორმულაში წარმოადგენს ისეთ უდიდეს რიცხვს, რომლისთვისაც P_k არის P_q-ს სუფიქსი.
P[1..m] ნიმუშის **პრეფიქს-ფუნქცია** ეწოდება ფუნქციას $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$, რომელიც განსაზღვრულია შემდეგნაირად: $\pi[q] = \max\{k : k < q \text{ და } P_k \sqsupset P_q\}$
სხვა სიტყვებით რომ ვთქვათ, π[q] არის P ნიმუშის იმ უდიდესი პრეფიქსის სიგრძე, რომელიც P_q-ს საკუთრივ სუფიქსს წარმოადგენს.

Algorithm 31: 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 32: 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)

ცხრილი 11.5

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 11.6: 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			
							a	b	a	b	a	c	a	

ტაბულა 11.7: სულ 15 შედარება

11.5 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 11.8

(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 B	E A	W O B	- B A	A A O	B B B	O A	U B	T	-	B	A	O	B	A	B	S

ცხრილი 11.9

(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:

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

წანაცვლების სიდიდეს ვინახავთ ცხრილში.

Algorithm 33: 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 11.10

11.6 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 11.11

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 11.11.
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 11.12

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.

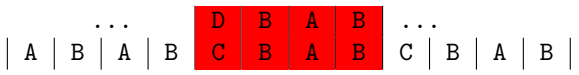
ამრიგად, "სდექ-სიმბოლოს ევრისტიკით" წანაცვლების სიდიდე d_1 ტოლია Table[c]-k, თუ ეს სიდიდე დადებითია, და უდრის 1-ს, თუ იგი არადადებითია.

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

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 11.13

ამ მაგალითში 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 11.14

ამ შემთხვევაში, ნიმუშის შესაძლო დასაშვები წანაცვლება იქნება:

ტაბულა 11.15

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 34: 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}).

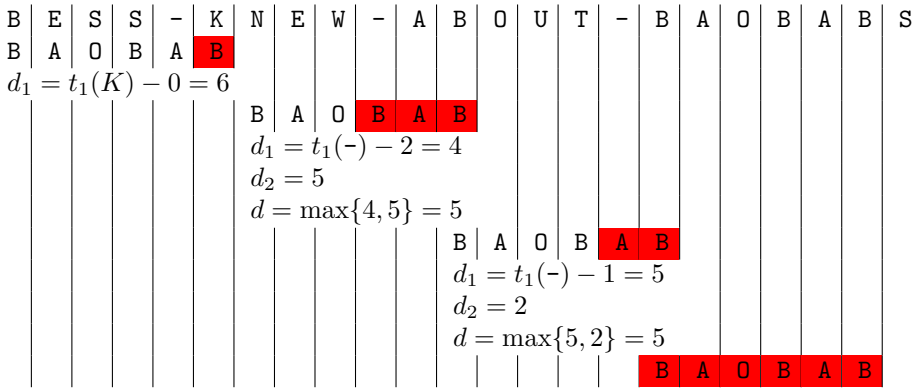
განვიხილოთ ლათინური ანბანის ასოებისგან და ხაზგასმის სიმბოლოებისგან შედგენილ ტექსტში: BESS-KNEW-ABOUT-BAOBABS, ნიმუშის - BAOBAB ძებნის ამოცანა.

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	$BAO\overline{BAB}$	2
2	$\overline{B}AOB\underline{AB}$	5
3	$\overline{B}AOBAB$	5
4	$\overline{B}A\underline{O}BAB$	5
5	$\overline{B}\underline{A}OBAB$	5

(b) საერთო სუფიქსების წანაცვლებები



(c) სულ 12 შედარება

ცაბულა 11.16

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).

11.7 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