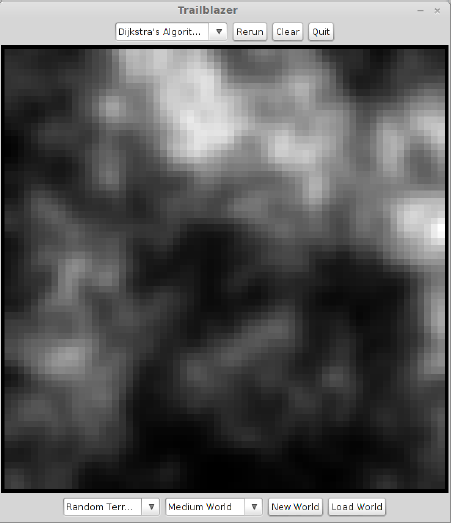
**Assignment: TraiBlazer**

ბოლო დავალებაში თქვენი მიზანი იქნება გამოიყენოთ თქვენი შესაძლებლობები და დაწეროთ სამი ალგორითმი: Dijkstra, A\* და Kruskal.

**The Program:**თქვენი დავალება არის დეიქსტრა და A\* ალგორითმების საშუალებით იპოვოთ უმოკლესი მანძილი ერთი წერტილიდან მეორემდე. ამის შემდეგ კი დაწეროთ Kruskal ის ალგორითმი შემთვევითი ლაბირინთი დაგენერირებისთვის.

იმისათვის რომ თვალსაჩინო გახდეს ალგორითმის მუშაობა, დავალებას მოყვება გრაფიკული პროგრამა რომლის საშუალებითაც ვიზუალურად დაინახავთ ალგორითმის მუშაობის პროცესს და შეძლებთ სამივე ალგორითმის გატესტვას. როცა გაუშვებთ პროგრამას თავიდან დაინახავთ ფანჯარას სადაც უკვე რანდომად იქნება დაგენერირებული რელიეფი, ისე როგორც სურათზეა ნაჩვენები.



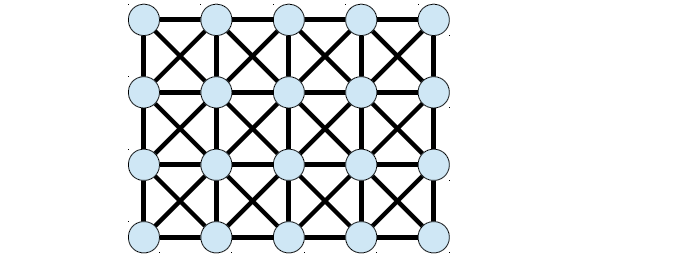
ღია ფერი ნიშნავს რომ მოცემული რელიები ზღვის დონიდან მაღლა მდებარეობს ხოლო მუქი ფერი კი პირიქით. შესაბამისად იქ სადაც მთებია შედარებით ღია ფერის უჯრები იქნება ხოლო ხრამები სადაცაა მუქი.  
თუ დააჭერთ ნებისმიერ 2 უჯრას სურათზე ავტომატურად გაეშვება ის ალგორითმი რომელიც არჩეულია drop down მენიუში (სურათზე დეიქსტაა არჩეული) რომ იპოვოს უმოკლესი გზა. სანამ უმოკლესი გზის პოვნის პროცესში იქნება სურათზე უჯრებს გააფერადებს მწვანედ, ყვითლად და ნაცრისფრად იმისდა მიხედვით ალგორითმი რომელ ფერს მიანიჭებს. როდესაც მანძილი ნაპოვნი იქნება პროგრამა გააფერადებს გზას და კონსოლში გამოიტანს რიცხვს რომელიც იქნება მთლიანი დანახარჯი.(ქვედა რელიეფიდან ზედა რელიეფზე ასვლა შედარებით უფრო დიდი ხარჯია ვიდრე პირიქით).

ჩვენ მიერ მოწოდებული გრაფიკული დიზაინი შეიცავს კიდევ რამდენიმე drop down მენიუს და buttonს რომლებიც საშუალებას მოგცემენ სხვადასხვა ზომის გრიდი დააგენერიროთ და უკეთ გატესტოთ თქვენი პრობლემა, რაშიც თქვენთვითონ მარტივად გაერკვევით.

როდესაც პირველად ჩართავთ პროგრამას ვინაიდან ზემოთ ხსენებული ალგორითმები არ არის დაწერილი, თუ შეეცდებით უმოკლესი მანძილის პოვნას ამოვარდება error message. ასევე არ იმუშავებს Random Maze Option რადგან Kruskal ის ალგორითმიც არ არის იმპლემენტირებული. შესაბამისად თავიდან არ გაგიკვირდეთ ეს მესიჯები და უბრალოდ გაეცანით ჩვენ მიერ მოწოდებულ გრაფიკულ დიზაინს რა option ები აქვს, შემდეგ კოდის წერას გაგიმარტივებთ.

**The Starter Code**

ჩვენ მოგაწოდეთ ბევრი კოდი რომელიც გამოგადგებათ ამ დავალებაში, აქ მოკლედ ავღწერთ თუ რას შეიცავს თითოეული ფაილი.  
**Trailblazer.h / .cpp** ეს არის ის ფაილები სადაც თქვენ მოგიწევთ დავალების 90% კოდის დაწერა (თუ 100% ის არა) სამივე გრაფის ალგორითმის რეალიზაცია ამ ფაილში იწერება.  
**TrailblazerTypes.h /.cpp** თქვენ არ მოგიწევთ ამ ფაილში ცვლილებების შეტანა, აქ უკვე მოცემული გაქვთ წინასწარ გამზადებული ტიპები მაგალითად ფერების, წიბოების და ადგილმდებარეობის შესანახად  
**TrailblazerPQueue.h** ეს არის PriorityQueues ოდნავ განსხვავებული ვერსია რომელსაც დამატებული აქ Decrease Key ოპერაცია, თქვენც შესაბამისად უნდა გამოიყენოთ ეს კლასი და ამ ფაილშიც ცვლილებები არ მოგიწევთ.  
**TrailblazerConstants.h** აღწერს კონსტანტებს რომლებიც გამოიყენება სხვადასხვა მოდულებში, თქვენ არ მოგიწევთ ამ კონსტანტებზე პირდაპირი წვდომა და არც შეცვლა შესაბამისად ამ ფაილსაც არ ეხებით.  
**TrailblazerCosts.h / .cpp**  აღწერს ფუნქციებს რომლებიც საჭიროა რო გამოთვალო ერთი რელიეიდან მეორეზე გადასვლის ხარჯი, ასევე ევრისტიკულ ფუნქციებს. თქვენ არ მოგიწევთ პირდაპირ გამოიყენოთ ან შეცვალოთ ეს ფაილი, თუმცა თუ extentions ს აპირებთ შეგიძლიათ.  
**TrailblazerGraphics.h / .cpp** ეს კლასი აკეთებს იმას რომ სწორად აჩენს გრაფიკულ ფანჯარას, უსმენს მაუსის ქლიქებს და ურთიერთობს ფანჯარასთან, თქვენ არ მოგიწევთ ამ კლასის შეცვლა  
**WorldGenerator.h / .cpp** აღწერს ფუნქციებს რომლებიც საჭიროა რელიეფის და ლაბირინთის დაგენერირებისთვის, თქვენ არ მოგიწევთ არც ამის შეცვლა.  
  
მართალია ბევრი კოდია აქ მაგრამ თქვენ მარტო Trailblazer.h / .cpp ში მოგიწევთ მუშაობა, დანარჩენი კოდები უბრალოდ დამხმარეა.  
  
**ნაბიჯი პირველი: Implement Dijkstra’s Algorithm**ამ ნაწილში თქვენი დავალებაა რომ დაწეროთ დეიქსტრას ალგორითმი რომელიც იპოვის უმოკლეს მანძილს ერთი წერტილიდან მეორემდე. თუ წარმოიდგენთ რელიეფს როგორც გრაფს (უფრო დეტალურად შემდეგ სექციაშია აღწერილი) მაშინ უმოკლესი მანძილის პოვნა საწყისი წერტილიდან საბოლოომდე ექვივალენტურია ამ გრაფში უმოკლესი მანძილის პოვნის.  
სამყაროს რომელსაც ჩვენ გაძლევთ წარმოდგენილია გრიდის სახით Grid<double> სადაც თითოეული double წარმოადგენს ზღვის დონიდან მანძილს ამ წერტილამდე. მიუხედავად იმისა რომ ჩვენ არ გაძლევთ გრიდს როგორც გრაფს შეგიძლიათ წარმოიდგინოთ რომ თითოეული წერტილი არის გრაფის წვერო რომელიც 8 მეზობელ წვეროსთან აკავშირებს (8 მეზობელ უჯრასთან) ისე როგორც სურათზეა ნაჩვენები.



რადგან სხვადასხვა სიმაღლის რელიეფიდან სხვადასხვა სიმაღლეზე ასასვლელად / ჩასასვლელად სხვადასხვა ძალისხმებაა საჭირო, თითოეულ წიბოს გრაფში აქ თავისი დანახარჯი, რომელიც აღნიშნავს ერთი რელიეფიდან მეორეზე გადასასვლელად რა ძალისხმევაა საჭირო. ჩვენ მოგაწვდით ფუნქციას რომელსაც გადასცემთ 2 მეზობელ წერტილს და გეტყვით ამ დანახარჯს/ძალისხმევას რიცხვებში. ეს ფუნქცია დააბრუნებს არაუარყოფით რიცხვს. ეს ფუნქცია ითვალისწინებს მოძრაობის მიმართულებსა (ცხადია დიაგონალის სიგრძე უფრო დიდია შესაბამისად უფრო მეტი დანახარჯია) და ასევე სიმაღლეებს ამ 2 წერტილის.   
თქვენი ამოცანა ამ ნაწილში არის დაწეროთ შემდეგი ფუნქცია.  
  
**Vector<Loc>  
shortestPath(Loc start, Loc end, Grid<double> & world, double constFunction(Loc one, Loc two, Grid<double> & world));**  
  
ეს ფუნქცია იღებს 4 პარამეტრს რომლებიც ქვემოთ არის აღწერილი.  
1) Loc start და loc end, Loc წარმოადგენს სტრუცტურას რომელსაც აქვს 2 ველი row და col, რომელიც აღნიშნავს გრიდში კორდინატს (Location \_ს ). ეს სტრუქტურა აღწერილია TrailblazerTypes.h ფაილში. Start და end პარამეტრები გადაეცემა ფუნქციას და თქვენი მიზანია ამ ორ წერტილს შორის იპოვოთ უმოკლესი გზა.  
2)Grid<double> word. წარმოადგენს გრიდს რომელშიც რელიეფის სიმაღლეებია შენახული.

3)constFunction(Loc one, Loc two, Grid<double> & world). ეს პარამეტრი არის ფუნქცია თუ პირველად ხედავთ ფუნქციის გადაცემას როგორც პარამეტრს მაშინ გადახედეთ handout ს (Functions and Parameters). ეს ფუნქცია იღებს ორ მეზობელ წერტილს პარამეტრად და თვითონ გრიდს და აბრუნებს დანახარს.  
თქვენი მიზანია დაწეროთ დეიქსტრას ალგორითმი რომელიც დათვლის უმოკლეს მანძილს ამ ორ წერტილს შორის და პასუხს დააბრუნებს Vector<Loc> ის სახით რომელშიც პირველი წერტილი იქნება start ბოლო კი end შუაშია რაც იქნება იქნება გზა.

მნიშვნელოვანი შენიშვნა: დეიქსტრას თქვენი იმპლემენტაცია უნდა იყოს ლექციაზე გარჩეული ალგორითმის მიხედვით, პრიორიტეტულ რიგში უნდა შეინახოთ node ები, ხოლო დამხმარე სტრუქტურის საშუალებით აღადგინოთ უმოკლესი გზა. ქვემოთ მოცემულია ლექციაზე გარჩეული ალგორითმის ფსევდო კოდი.

დეიქსტრას ალგორითმის ფსევდო-კოდი (startNode-დან endNode-მდე):

* გააფერადეთ ყველა წვერო ნაცრისფრად
* გააფერადეთ startNode ყვითლად
* მიანიჭეთ startNode-ს სავარაუდო მანძილი (პრიორიტეტი) 0
* ჩაამატეთ startNode განსახილველი node-ების priority queue-ში
* სანამ განსახილველი წვეროების რიგში ჯერ კიდევ არ წვეროები:
  + ამოიღეთ უმცირესი პრიორეიტეტის წვერო (curr) რიგიდან
  + გააფერადეთ curr მწვანედ (ამ ეტაპზე ვიცით რომ სავარაუდო მანძილი ამ წვერომდე ნამდვილად უმცირესი მანძილია)
  + თუ curr არის endNode, მაშინ გზა ნაპოვნია და ალგორითმი მორჩა
  + ყოველი curr-ის მეზობელი წვერო v-სთვის (L სიგრძის წიბოთი):
    - თუ v ნაცრისფერია:
      * გააფერადეთ v ყვითლად
      * მიანიჭეთ v-ს სავარაუდო მანძილი dist(curr) + L
      * მიანიჭეთ v-ს მშობელ წვეროდ curr
      * ჩასვით v განსახილველი წვეროების რიგში შესაბამისი პრიორიტეტით
    - თუ v უკვე ყვითელია და v-მდე არსებული სავარაუდო მანძილი dist(curr) + L -ზე მეტია:
      * მიანიჭეთ v-ს სავარაუდო მანძილი dist(curr) + L
      * მიანიჭეთ v-ს მშობელ წვეროდ curr
      * შეცვალეთ v-ს პრიორიტეტი განსახილველი წვეროების რიგში ახალი სავარაუდო მანძილის მიხედვით.

ამ ფსევდო კოდის ფარგლებში შეგიძლიათ შეგიძლიათ ეს ფუნქცია როგორც გინდათ ისე შეასრულოთ და რა მონაცემთა სტრუქტურებიც გინდათ ისინი გამოიყენოთ. სანამ ამ ფუნქციის იმპლემენტაციას დაიწყებთ, სასარგებლო იქნება თუ შემდეგ საკითხებზე დაფიქრდებით:

• როგორ შეინახავთ თითოეული წვეროს მშობელ წვეროს?

• როგორ შეინახავთ რომელი წვერო არის ნაცრისფერი/ყვთელი/მწვანე?

• როგორ შეინახავთ სავარაუდო (და შემდეგ საბოლოო) მანძილებს თითოეულ წვერომდე?

თქვენი იმპლემენტაციის დეტალური ოპტიმიზაცია არ არის საჭირო, მაგრამ ეცადეთ ისეთი მონაცემთა სტრუქტურები შეარჩიოთ ხოლმე რომ შესაბამისი ამოცანა სწრაფად შეასრულონ. ამ ამოცანაში თქვენი შეფასების ქულის სტილის ნაწილი მონაცემთა სტრუქტურების არჩევანზე იქნება დამოკიდებული, ასე რომ კარგი იქნება თუ თქვენს არჩევანს მოკლედ დაასაბუთებთ ხოლმე კომენტარებით. მაგალითად რომელი წვეროები არის მწვანე იმის დამახსოვრება მწვანე წვეროების ვექტორში ჩაყრით არ იქნება კარგი იდეა. ასეთი იმპლემენტაცია არც მარტივი არ იქნება და არც სწრაფი. გახსენებთ რომ სავარაუდოდ ერთ-ერთ მონაცემთა სტრუქტურად TrailblazerPQueue გამოგადგებათ რადგან ამ სტრუქტურად decreaseKey ოპერაცია აქვს.

დეიქსტრას ალგორითმი უმოკლეს მანძილს არა მხოლოდ დანიშნულების ადგილამდე, არამედ ბევრ სხვა წვერომდეც ითვლის. ეს ბუნებრივია და დექისტრას ალგორითმი ამის გარეშე ვერ იმუშავებს. იმისთვის რომ ვიზუალურად დაინახოთ რას აკეთებს დეიქსტრას ალგორითმი TrailblazerGraphics.h-ში მოცემული გაქვთ გამზადებული ფუნქციები, რომლებიც საშუალებას გაძლებთ მსოფლიოს უჯრები გააფერადოთ:

**void colorCell(Grid<double>& world, Loc loc, Color color);**

ეს ფუნქცია იღებს სამ პარამეტრს – მსოფლიოს აღმწერ ცხრილს (grid), მსოფლიოს ერთ-ერთი უჯრის მდებარეობას და ფერს, შემდეგ კი აფერადებს მოცემულ უჯრას ამ ფერით (Color ტიპის ცვლადი enum არის და მისი განმარტება Trailblazergraphics.h-ში არის მოცემული. Enum-ების შესახებ შეგიძლიათ კურსის წიგნის 1.5 და 1.7 თავებში წაიკითხოთ). თქვენი shortestPath ფუნქცია colorCell ფუნქციას შემდეგ შემთხვევებში უნდა იძახებდეს:

• როდესაც წვეროს (node) განსახილველი წვეროების რიგში პირველად ამატებთ, ეს უჯრა ყვიტლად უნდა გააფერადოთ

• რედესაც უჯრას (წვეროს) განსახილველი წვეროების სიიდან იღებთ და განიხილავთ, ეს უჯრა მწვანედ უნდა გააფერადოთ

colorCell ფუნქციას მხოლოდ ეკრანზე ფერების გამოტანა ეხება და არა მონცებებს შენახვა, რომელი უჯრა არის ყვითელი/მწვანე თქვენ თვიტონ უნდა შეინახოთ.

პროგრამის განმავლობაში ეს ფუნქცია საშუალებას მოგცემთ თვალყური ადევნოთ ზუსტად როგორ მუშაობს ალგორითმი (თან სახალისო საყურებელია!). დავალებს შემდეგ ნაწილში შეგიძლიათ ეს კოდი დეიქსტრას და A\* ალგორიტმების შესადარებლად გამოიყენოთ.

სანამ შემდეგ სექციაზე გადახვალთ, გირჩევთ ეს ალგორითმი ბევრ სხვადასხვა ტესტზე გამოსცადოთ. შეგიძლიათ თქვენი პროგრამის ნაპოვნი გზები მოცემულ სანიმუშო გზებს შეადაროთ. ყურადღება მიაქციეთ რომ შეიძლება რამდენიმე ოპტიმალური გზა იყოს და შესაბამისად თქვენი გზა შეიძლება ზუსტად არ დაემთხვეს ნიმუშს. თუმცა თქვენი ალგორითმის დაბრუნებული გზის სიგრძე იგივე უნდა იყოს რაც ოპტიმალური გზის სიგრძე. ამის მარტივად შესამჩნებად, გამზადებული კოდი ეკრანზე გამოგიტანთ თზვენი პროგრამის მიერ ნაპოვნი გზის ჯამურ სიგრძეს.

ამ პროგრამის დასაწერად დაახლოებით 75 ხაზი კოდი (კომენტარების ჩათვლით) საკმარისი უნდა იყოს. ამაზე ბევრად მეტის წერა თუ დაგჭირდათ სავარაუდოდ კოდს უნდა გადახედოთ და უკეთეს სტილით/დეკომოზიციით დაწეროთ.

**ნაბიჯი ორი: A\* ალგორითმის იმპლემენტაცია**

ამ დავალების შემდეგ (ამ) ნაწილში უმოკლესი გზის პოვნის პროგრამა ისე უნდა შეცვალოთ რომ მას დეიქსტრას მიხედვითაც შეეძლოს მუშოაბა და A\* ალგორითმის მიხედვითაც. შეცვალეთ shortestPath ფუნქციის დეკლარაცია (პროტოტიპი) შემდეგნაირად:

Vector<Loc> shortestPath( Loc start, Loc end, Grid<double>& world,   
double costFunction( Loc one, Loc two,  
Grid<double>& world),   
double heuristic(Loc start, Loc end, Grid<double>& world));

(ამ პროტოტიპის შეცვლა ორ ადგილას მოგიწევთ. ერთხელ Trailblazer.cpp-ში და მეორედ Trailblazer.h-ში. თუ ორივე ადგილას არ შეცვლით პროტოტიპს, შეცდომა მოხდება)

ახლა თქვენი ფუნქცია უნდა იღებდეს ევრისტიკის გამომთვლელ ფუნქციას დამატებით პარამეტრად. ეს ევრისტიკა (heuristic) იღებს სამ პარამეტრს: საწყის უჯრას, საბოლოო უჯრას და მსოფლიოს აღწერას და აბრუნებს სავარაუდო მანძილს საწყისი მდგომარეობიდან საბოლოომდე. შეგიძლიათ ჩათვალოთ რომ ეს ევრისტიკა ყოველთვის დასაშვები (admissible) იქნება, ანუ არასოდეს არ დააბრუნებს ნამდვილ მანძილზე მეტს. ამ დამატებითი პარამეტრის გამოყენებით, შეცვალეთ თქვენი ფუნქცია ისე, რომ A\* ალგორითმს ასრულებდეს და არა დეიქსტრას ალგორითმს. A\* ალგორითმის ფსევდოკოდი ამ დავალებაში ქვემოთ არის მოცემული.

მიუხედავად იმისა რომ თქვენი ფუნქცია ახლა ყოველთვის იღებს ევრისტიკას პარამეტრად, მაინც შეგიძლიათ ის გამოიყენოთ დეიქსტრას ალგორითმის შესასრულებლად. ამისათვის უბრალოდ გაიხსენეთ რომ თუ ევრისტიკა ყოველთვის 0-ს აბრუნებს სავარაუდო მანძილად მაშინ A\* ალგორითმი დეიქსტრას ალგორითმს ემთხვევა. გამზადებული კოდი ისე არის მოწყობილი, რომ როდესაც ფუნქციის პროტოტიპს გადააკეთებთ, თუ დეიქსტრას ალგორითმის გაშვებას აირჩევთ, პროტოტიპის უკან გადაკეთება არ იქნება საჭირო. ამის მაგივრად გამშვები კოდი ავტომატურად გადასცემს თქვენს პროგრამას ევრისტიკად ისეთ ფუნქციას, რომელიც ყოველთვის 0-ს აბრუნებს.

თუ A\*-ის გაშვებას აირჩევთ, გამშვები პროგრამა წინასწარ გამზადებულ ევრისტიკას გადასცემს თქვენს ფუნქციას. წინა ნაწილიდან მოყოლებული, თქვენი კოდი უკვე აფერადებს რიგში ჩამატებულ უჯრებს. ამიტომ, როგორც კი A\*-ს გაუშვებთ ეკრანზე დაინახავთ რამდენი უჯრის განხილვა დასჭირდება A\*-ს ძებნისას.

**ამ ნაწილის შესასრულებლად კოდში ბევრი ცვლილების შეტანა არ მოგიწევთ.** დაახლოებით ხუთი ხაზი კოდის გადაკეთება შეიძლება საკმარისი იყოს დეიქსტრადან A\*-ის მისაღებად, თუმცა თქვენი იმპლემენტაციის მიხედვით შეიძლება მეტის შეცვლაც მოგიწიოთ. შეგიძლიათ ქვემოთ მოცემული კოდი ნახოთ და შეადაროთ დეიქსტრას ფსევდო-კოდს.

A\*-ის ფსევდო-კოდი (startNode-დან endNode-მდე h ევრისტიკით):

* გააფერადეთ ყველა წვერო ნაცრისფრად
* გააფერადეთ startNode ყვითლად
* მიანიჭეთ startNode-ს სავარაუდო მანძილი 0
* ჩაამატეთ startNode განსახილველი node-ების priority queue-ში პრიორიტეტით h(startNode, endNode)
* სანამ განსახილველი წვეროების რიგში ჯერ კიდევ არ წვეროები:
  + ამოიღეთ უმცირესი პრიორეიტეტის წვერო (curr) რიგიდან
  + გააფერადეთ curr მწვანედ (თუ ევრისტიკა მონოტონურია, მაშინ ამ ეტაპზე ვიცით რომ სავარაუდო მანძილი ამ წვერომდე ნამდვილად უმცირესი მანძილია)
  + თუ curr არის endNode, მაშინ გზა ნაპოვნია და ალგორითმი მორჩა
  + ყოველი curr-ის მეზობელი წვერო v-სთვის (L სიგრძის წიბოთი):
    - თუ v ნაცრისფერია:
      * გააფერადეთ v ყვითლად
      * მიანიჭეთ v-ს სავარაუდო მანძილი dist(curr) + L
      * მიანიჭეთ v-ს მშობელ წვეროდ curr
      * ჩასვით v განსახილველი წვეროების რიგში მოცემული პრიორიტეტით: dist(curr) + L + h(v, endNode)
    - თუ v უკვე ყვითელია და v-მდე არსებული სავარაუდო მანძილი dist(curr) + L -ზე მეტია:
      * მიანიჭეთ v-ს სავარაუდო მანძილი dist(curr) + L
      * მიანიჭეთ v-ს მშობელ წვეროდ curr
      * შეცვალეთ v-ს პრიორიტეტი განსახილველი წვეროების რიგში შემდეგ მნიშვნელობაზე: dist(curr) + L + h(v, endNode)

დეიქსტრას და A\* ალგორითმების შესადარებლად შეგიძლიათ “rerun” ღილაკი გამოიყენოთ. შეგიძლიათ ჯერ დეიქსტრა გაუშვათ, შემდეგ მენიუდან A\* აირჩიოთ და ამ ღილაკის დაჭერით ახლა A\* გაუშვათ რომ შედეგები იგივე რუკაზე შეადაროთ. სავარაუდოდ A\* ალგორითმი ბევრად უფრო ეფექტური აღმოჩნდება.

როგორც დეიქსტრას გამოცდისას, აქაც შეამოწმეთ რომ თქვენი ალგორითმი სწორი სიგრძის პასუხს აბრუნებს. როგორც ადრე, ახლაც შეიძლება ოპტიმალური გზა ერთადერთი არ იყოს, მაგრამ ყველა მათგანის სიგრძე აუცილებლად უნდა ემთხვეოდეს. რადგან მოცემული ევრისტიკა დასაშვები (admissible) იქნება, თქვენი პროგრამა ერთ-ერთ ოპტიმალურ გზას უნდა აბრუნებდეს.

ასევე, მოცემულ ამოხსნას ისიც შეადარეთ რომელი ადგილები განიხილა პროგრამამ. თუ განხილულ ადგილებში დიდი განსხვავებაა შეიძლება თქვენი დაწერილი პროგრამა არასწორად მუშაობს.

შენიშვნა: როდესაც ევრისტიკა მხოლოდ დასაშვებია და არა მონოტონური A\*-ში შეიძლება ხანდახან მწვანე წვერომდე გზა გაუმჯობესდეს. როდესაც ეს ხდება, ეს მწვანე წვერო თავიდან უნდა ჩასვათ ხოლმე განსახილველი (ყვითელი) წვეროების რიგში შესაბამისი ახალი პრიორიტეტით. ამ დავალებაში ამის გაკეთება არ იქნება საჭირო და ამიტომ ფსევდოკოდში არც არის გათვალისწინებული შესაბამისი ოპერაციები.

იმისათვის, რომ გატესტოთ თქვენი პროგრამა მუშაობს თუ არა სწორად, შეეცადეთ შეადაროთ თქვენი პროგრამის შედეგი, ჩვენი ამონახსნის პასუხს. თქვენს ამონახსნში შეიძლება მიიღოთ ჩვენი ამონახსნისგან ოდნავ განსხვავებული უმოკლესი გზა, მაგრამ გზის ჯამური ღირებულება არ უნდა განსხვავდებოდეს. თქვენ ასევე უნდა მიაქციოთ ყურადღება რომელი ლოკაციები მოინახულა იმპლემენტაციამ, თუ მათ შორის დიდი განსხვავება იქნება, ეს შეიძლება ნიშნავდეს იმას, რომ პროგრამაში ბაგი გაქვთ.

დამატებით, რადდგან ჩვენი მოცემული ევრისტიკა არის დასაშვები ევრისტიკა, ეს იმას ნიშნავს, რომ თქვენმა A\* ძებნის ალგორითმა ყოველთვის უნდა დააბრუნოს იგივე ღირებულების უმოკლესი გზა, რასაც დეიქსტრას ალგორითმი აბრუნებს. წინააღმდეგ შემთხვევაში, რაღაც ბაგი გაქვთ თქვენს ამონახსნში.

**მესამე ნაწილი: დააიმპლემენტირეთ კრასკალის ალგორითმი**

თქვენ ახლა გაქვთ კარგი გზის პოვნის ალგორითმი, მაგრამ როგორ იმუშავებს იმ შემთხვევაში თუ ცდილობთ ლაბირითიდან გამოიკვლიოთ გზა? თქვენი საბოლოო დავალება არის, დაწეროთ ლაბირითის ამგები, კრასკალის ალგორითმის გამოყენებით. როგორც გახსოვთ, კრასკალის ალგორითმი გამოიყენება მოცემულ გრაფში უმოკლესი დამფარავი ხის ასაგებად. კრასკალის ალგორითმის ფსევდოკოდი მოცემული არის ქვემოთ:

Kruskal's algorithm on a graph:

• Place each node into its own cluster.

• Insert all edges in the graph into a priority queue.

• While there are two or more clusters remaining:

• Dequeue an edge *e* from the priority queue.

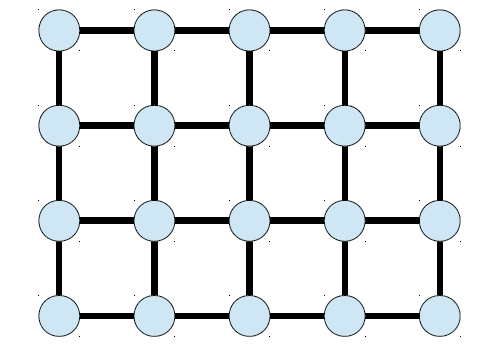
• If the endpoints of *e* are not in the same cluster:

• Merge the clusters containing the endpoints of *e*.

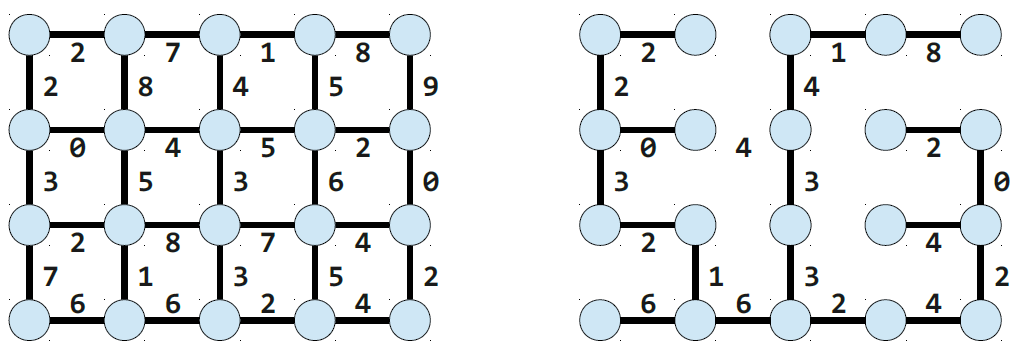
• Add *e* to the resulting spanning tree.

• Return the spanning tree formed this way.

ახლა გამოვიყენოთ კრასკალის ალგორითმი ლაბირინთის ასაგებად. წარმოიდგინეთ, რომ მოცემული გაქვთ grid გრაფი ისე, როგორც სურათზეა წარმოდგენილი:

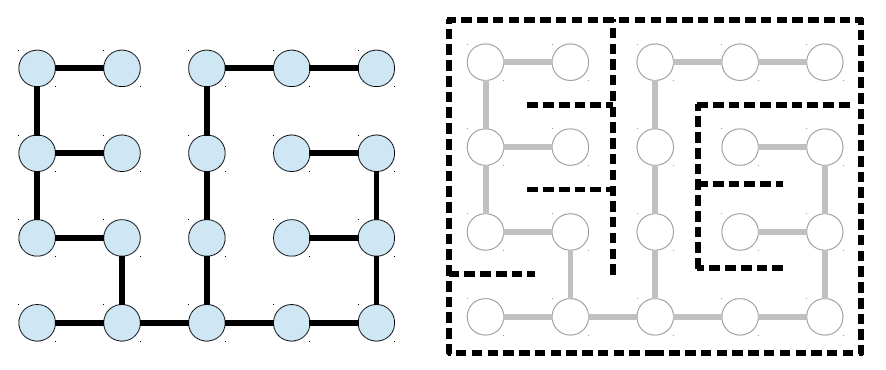


თუ თქვენ მის წიბოებს მიანიჭებთ რენდომ წონებს და გაუშვებთ კრასკალის ალგორითმს დაგრჩებათ მინიმალური დამფარავი ხე. ასევე ნებისმიერ 2 წვეროს შორის იარსებებს ერთადერთი გზა. მაგალითისთვის, როგორც სურათზეა ნაჩვენები, თუ მივანიჭებდით ამ წონებს შედეგი გვექნებოდა ასეთი:



ყურადღება მიაქციეთ და გაიაზრეთ, რატომ მივიღეთ ეს შედეგი.

თქვენ შეგიძლიათ ეს ხე წარმოიდგინოთ, როგორც ლაბირინთი. როგორც წესი, როდესაც ვხატავთ ლაბირინთს, ხაზები წარმოადგენენ კედლებს, ხოლო სიცარიელე ნიშნავს გზას. ჩვენ ხეში კი პირიქით, სადაც გადის ხაზი ის ნიშნავს, რომ შეგვიძლია გავლა, სიცარიელე კი დაბრკოლებას წარმოადგენს, სადაც ვერ გავივლით. თქვენ ზემოთ მოცემული ხე შეგიძლიათ წარმოსახოთ ლაბირინთად, თუ სიცარიელეებში გაავლებთ ხაზებს, როგორც სურათზეა ნაჩვენები:



თქვენი დავალება ამ ნაწილში არის, გამოიყენოთ რენდომ ვერსია კრასკალის ალგორითმის, რათა ააგოთ რენდომ ლაბირინთი. უფრო ზუსტად, თქვენი დავალება არის დაწეროთ შემდეგი ფუნქცია:

**Set<Edge> createMaze(int numRows, int numCols);**

ამ ფუნქციას გადმოეცემა ორი რიცხვი, რომლებიც აღნიშნავს რიგების რაოდენობას და სვეტების რაოდენობას ლაბირინთში. შემდეგ კი უნდა ააგოთ ამ სიდიდის რენდომ ლაბირინთი. ფუნქციამ უნდა დააბრუნოს Set<Edge> სადაც edge წარმოადგენს დამფარავ ხეში შემავალ წიბოებს. Edge ტიპი (აღწერილი TrailblazerTypes.h-ში) არის struct, რომელიც ინახავს ორ Loc-ს, რომელიც წარმოადგენს წიბოს ორ დაბოლოებას.

.

გაითვალისწინეთ, რომ დავალების პირველი ნაწილისგან განსხვავებით, აქ მხოლოდ გადმოგეცემათ გრაფის ზომები და არა თავად გრაფი. თქვენ უნდა შექმნათ grid გრაფი, წიბოებს მიანიჭოთ რენდომ წონები (იმისათვის რომ გქონდეთ წონების კარგი განაწილება გირჩევთ გამოიყენოთ randomReal(0, 1) წონების მისანიჭებლად).

სანამ კოდის წერას დაიწყებდეთ, გირჩევდით დაფიქრდეთ შემდეგ კითხვებზე:

• როგორ შეინახავთ წვეროებს და წიბოებს გრაფში?

• როგორ შეინახავთ რომელი წვეროები არის რომელ კლასტერში?

• როგორ განსაზღვრავთ წვერო რომელ კლასტერს ეკუთვნის?

• როგორ გააერთიანებთ 2 კლასტერს?

გაითვალიწინეთ, რომ დავალების ამ ნაწილში, თქვენ მოგიწევთ ზუსტად ააგოთ ყველა წიბო grid გრაფში. თქვენ, შესაძლოა გინდოდეთ დახატოთ სურათი, რათა გაარკვიოთ როგორ აკეთებთ ამას. ჩვენი საწყისი კოდი დაწერილია ისე, რომ იძახებს თქვენს ფუნქციას, ააგოს რენდომ ლაბირინთ, როდესაც თქვენ იყენებთ

New World ღილაკს “Random Maze”-ით მონიშნულს. როდესაც ააგებთ ლაბირინთს, თქვენ შეგიძლიათ გაუშვათ Dijkstra's ან A\* იმპლემენტაცია, რათა იპოვოთ გზა ლაბირინთის ორ წერტილს შორის. თქვენი უმოკლესი გზის ფუნქცია არგუმენტებად იღებს ორ ფუნქციას - ერთი ორ წერტილს შორის წიბოს წონას აბრუნებს. მეორე კი ევრისტიკას ითვლის. როდესაც თქვენ გამოიყენებთ ჩვენს პროგრამას უმოკლესი გზის საპოვნელად ლაბირინთში, ჩვენ გადმოვცემთ shortestPath მეთოდს ფუნქციას, რომელიც ითვლის ღირებულებებს შემდეგნაირად:

• კედელს შორის გავლის ღირებულება იქნება უსასრულობა.

• დიაგონალზე გადაადგილების ღირებულება იქნება უსასრულობა.

**Advice, Tips and Tricks**

იხილეთ დავალების ინგლისურ ვერსიაში.

**Possible Extensions**

იხილეთ დავალების ინგლისურ ვერსიაში.