ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი

ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერულ მეცნიერებათა დეპარტამენტი

ელდარ ბოგდანოვი

სამაგისტრო ნაშრომი:

უმოკლესი გზების განსაზღვრა ევკლიდურ გრაფებში

ნაშრომი შესრულებულია კომპიუტერულ მეცნიერებათა მაგისტრის აკადემიური ხარისხის მოსაპოვებლად.

ხელმძღვანელი: პროფესორი კობა გელაშვილი

თბილისი

2013

სარჩევი

ანოტაცია

დღესდღეობით ალგორითმების ერთ-ერთ ფართო კლასს წარმოადგენს გრაფებზე ალგორითმები. გრაფების მეშვეობით ხდება უამრავი ტიპის მიმართების და პროცესის მოდელირება ინფორმაციულ, ფიზიკურ, ბიოლოგიურ, სოციალურ სისტემებში. გრაფების გამორჩეულ სახეობას წარმოადგენს ევკლიდური გრაფები, რომელთა გამოყენების ერთ-ერთი ძირითადი სფერო საგზაო სისტემების მოდელირებაა. ასეთ სისტემებში მუდმივად დგება პუნქტების წყვილებს შორის უმოკლესი მარშრუტის მოძებნის ამოცანები, რომლებისთვისაც გრაფთა თეორიაში შესაბამისი ალგორითმები არსებობს.

ნაშრომის მიზანია შემთხვევითი ევკლიდური გრაფების ორი სახეობის - ბერნულის და გეომეტრიული გრაფების სწრაფი გენერაციის ხერხების მოძიება, უმოკლესი გზის მოძებნის არსებული კლასიკური ალგორითმების ეფექტური იმპლემენტაცია და მათი წარმადობის შედარება ამ ტიპის გრაფებზე სხვადასხვა განზომილების სივრცეებში. ასევე გამოკვლეული იყო ახალი ევრისტიული მეთოდის წარმადობა და ოპტიმალურობა.

Abstract

Nowadays graph algorithms constitute a broad class of modern algorithms. Graphs can be applied to model relations and processes in informational, physical, biological and social systems. One notable type of graphs is Euclidean graph, which is often used to represent road networks. A frequent task in such networks is determining the shortest route between two sites. There are special Graph Theory algorithms which solve this problem.

The goals of this thesis are: to research the means of efficient generation of graphs according to two random Euclidean graph models – Bernoulli Random Graphs and Random Plane Networks; to implement several classical shortest path algorithms and to compare their performance on the generated graphs in different space dimensions. Also, a new heuristical method was suggested and its performance and optimality studied.

შესავალი

დღესდღეობით ალგორითმების ერთ-ერთ ფართო კლასს წარმოადგენს გრაფებზე ალგორითმები. გრაფების მეშვეობით ხდება უამრავი ტიპის მიმართების და პროცესის მოდელირება ინფორმაციულ, ფიზიკურ, ბიოლოგიურ, სოციალურ სისტემებში.

გრაფების გამორჩეულ სახეობას წარმოადგენს ევკლიდური (ასევე ცნობილი როგორც „გეომეტრიული“) გრაფები. მათი გამოყენების ერთ-ერთი ძირითადი სფერო საგზაო სისტემების მოდელირებაა. აქ იგულისხმება ნებისმიერი მაშტაბის საგზაო სისტემები, დაწყებული ქვეყნის დასახლებულ პუნქტებს შორის საგზაო მიმოსვლით და დამთავრებული ერთი ქალაქის ფარგლებში არსებული გზებით.

ძირითადი საკითხი, რომელიც დგება საგზაო სისტემებთან მუშაობისას არის მარშრუტების მოძებნა პუნქტების წყვილებს შორის. ხშირად საძიებელი ხდება უმოკლესი მარშრუტები. ეს ამოცანა რეგულარულად ისმება მაგალითად ელექტრონული რუკების გამოყენებისას და მისი სწრაფი გადაწყვეტა ერთ-ერთ პრიორიტეტულ საკითხს წარმოადგენს.

გრაფში უმოკლესი გზის მოძებნის ამოცანა მე-20 საუკუნის 50-იანი წლებიდან ინტენსიური კვლევის საგანს წარმოადგენს. 1955 წელს შიმბელის მიერ იყო გამოქვეყნებული ამ ამოცანის გადამწყვეტი პირველი ალგორითმი, რომელსაც 1956 წელს ფორდის და შემდეგ 1958 წელს ბელმანის ალგორითმი მოჰყვა. 1959 წელს გამოქვეყნდა დეიქსტრას ალგორითმი, რომელიც დღესაც უმოკლესი გზის პოვნის კლასიკურ ალგორითმს წარმოადგენს. შემდეგ მოახდინეს ამ ალგორითმის მოდიფიკაცია ხალვათი გრაფებისთვის ჯერ ორობითი გროვების გამოყენებით, შემდეგ კი ფიბონაჩის გროვების გამოყენებით (1984). 1986 წელს გამოქვეყნდა სეჯვიკის და ვიტერის A* ძებნაზე დაფუძნებული ალგორითმი, რომელიც ევრისტიულად ასწრაფებს უმოკლესი გზის მოძებნას ევკლიდურ გრაფში. ამ საკითხებზე მუშაობა შემდეგაც გრძელდებოდა და ამ მიმართულებით ერთ-ერთი ბოლო წინსვლა 2004 წელს Microsoft Research-ის მიერ გამოქვეყნებული Reach ევრისტიკა იყო.

ამ ნაშრომის მიზანია უმოკლესი გზის პოვნის არსებული კლასიკური და ევრისტიული ალგორითმების სწრაფქმედების შედარებითი ანალიზი სხვადასხვა სახის შემთხვევით

ევკლიდურ გრაფებზე. ასევე იყო გამოკვლეული უმოკლესთან მიახლოებული მარშრუტის დადგენის ახალი ევრისტიული მეთოდები.

ნაშრომში ძირითადი აქცენტი გაკეთებულია სიბრტყეზე განლაგებულ ევკლიდურ გრაფებზე, როგორც ევკლიდური გრაფების აქტუალურ მოდელზე. ასევე არის გამოკვლეული ზოგიერთი ალგორითმის და ევრისტიკის მუშაობა უფრო მაღალი განზომილების ევკლიდურ გრაფებზე.

ნაშრომი შედგება 5 თავისგან. პირველ თავში მოყვანილია გრაფთა თეორიის ძირითადი ცნებები, რომლებიც გამოყენებული იქნება ნაშრომში. მეორე თავში ხდება იმ კლასიკური ალგორითმების მიმოხილვა, რომლების იმპლემენტაცია და შედარებითი ანალიზი შესრულდა ნაშრომის ფარგლებში. მესამე თავი ეთმობა ახალი ევრისტიული MinimumAngleDFS მეთოდის მიმოხილვას, რომელიც გამოიყენება უმოკლესი მარშრუტის სიგრძეზე გარკვეული ზედა ზღვრის დასადგენად. მეოთხე თავში შეჯამებულია იმპლემენტაციის თავისებურებები, რომელიც თან ახლდა ჩატარებულ მოდელირებას. მეხუთე თავში შეფასებულია მოდელირების შედეგები.

თავი 1. ძირითადი ცნებები

გრაფი (Graph). **G**=(**V**, **E**) გრაფი წარმოადგენს წვეროების **V** სიმრავლის და წიბოების **E** სიმრავლის წყვილს. ყოველი წიბო წარმოადგენს წვეროების დალაგებულ ან დაულაგებელ წყვილს. წიბოები მიმართულია, თუ **E** სიმრავლის ელემენტები დალაგებული წყვილებია, წინააღმდეგ შემთხვევაში - ორმხრივი. გრაფს ეწოდება წონიანი, თუ ყოველ წიბოსთან ასოცირებულია რიცხვი - ამ წიბოს წონა. წინააღმდეგ შემთხვევაში გრაფს უწონო ეწოდება.

მკვრივი (Dense) და ხალვათი (Sparse) გრაფები. გრაფს ეწოდება მკვრივი, თუ მისი წიბოების რაოდენობა |E| იგივე რიგისაა, რაც წიბოების მაქსიმალური შესაძლო რაოდენობა, ანუ $|E| \sim |V^2|$. წინააღმდეგ შემთხვევაში გრაფს ხალვათი ეწოდება.

ევკლიდური გრაფი (Euclidean Graph). **d**-განზომილებიან სივრცეში განლაგებული ევკლიდური გრაფის ყოველ წვეროსთან ასოცირებულია **d**-განზომილებიანი სივრცის წერტილი $(\mathbf{x}^1, \ldots, \mathbf{x}^d)$. გრაფში არსებული ყოველი წიბოს წონა შესაბამის წვეროებს შორის ევკლიდური მანძილის ტოლია.

ბერნულის შემთხვევითი გრაფი (Bernoulli Random Graph) [1]. **d**-განზომილებიან სივრცეში განლაგებული ბერნულის შემთხვევითი **G(n, p)** გრაფი არის **n**-წვეროიანი ევკლიდური გრაფი, რომლის წვეროებს **d**-განზომილებიანი სივრციდან აღებული შემთხვევითი წერტილები შეესაბამება, ხოლო ყოველი შესაძლო წიბოს არსებობის ალბათობა დამოუკიდებელია და ტოლია **p**-სი.

შემთხვევითი გეომეტრიული გრაფი (Random Plane Network) [2]. **d**-განზომილებიან სივრცეში განლაგებული შემთხვევითი გეომეტრიული **G(n, r)** გრაფი წარმოადგენს **n**-წვეროიან ევკლიდურ გრაფს, რომლის წვეროებს **d**-განზომილებიანი სივრციდან აღებული შემთხვევითი წერტილები შეესაბამება, ხოლო ყოველი (U, V) წიბო არსებობს მაშინ და მხოლოდ მაშინ, თუ ევკლიდური მანძილი U და V წვეროებს შორის არ აღემატება **r**-ს.

გზა გრაფის წვეროებს შორის (Path). **G=(V, E)** წონიან გრაფში **a** და **b** წვეროებს შორის გზა წარმოადგენს წვეროების ისეთ $v_1, v_2, \ldots, v_K$ მიმდევრობას, რომ $v_1 = \mathbf{a}$, $v_k = \mathbf{b}$ და ყოველი i<K-

სთვის **E** შეიცავს (v_i, v_{i+1}) წიბოს. ასეთი გზის სიგრძე **L(v_1, …, v_K)** ეწოდება $\sum_{i=1}^{K-1} l(v_i, v_{i+1})$ ჯამს, სადაც **l(u, v)** არის u და v წვეროების დამაკავშირებელი წიბოს წონა.

უმოკლესი გზა გრაფის წვეროებს შორის (Shortest Path). a და **b** წვეროებს შორის უმოკლესი გზის სიგრძე **D(a, b)** არის მინიმალური რიცხვი **a** და **b** წვეროების შემაერთებელი ყველა შესაძლო გზის სიგრძეებს შორის. თუ ასეთი გზა არ არსებობს, უმოკლესი გზის სიგრძე განმარტებული არ არის. თვითონ უმოკლესი გზა **a** და **b** წვეროებს შორის არის ნებისმიერი ისეთი გზა ამ წვეროებს შორის, რომლის სიგრძე ტოლია **D(a, b)**-სი.

ორობითი გროვა (Binary Heap) [3]. ორობითი გროვა წარმოადგენს მონაცემთა სტრუქტურას, რომელიც სრული ორობითი ხით წარმოიდგინება და შეუძლია შემდეგი სამი ფუნქციის შესრულება:

- სიმრავლეში ელემენტის დამატება $O(\log_2 N)$ ალოგორითმული სირთულით, სადაც N სიმრავლეში ელემენტების რაოდენობაა.
- სიმრავლის ელემენტის მნიშვნელობის შემცირება $O(\log_2 N)$ სირთულით.
- სიმრავლიდან მინიმალური ელემენტის ამოშლა $O(\log_2 N)$ სირთულით.
- სიმრავლის მინიმალური ელემენტის მოძიება $O(1)$ სირთულით.

ფიბონაჩის გროვა (Fibonacci Heap) [4]**.** ორობითი გროვის მსგავსი სტრუქტურა, რომელიც სიმრავლის მინიმალური ელემენტის ძიებას $O(1)$ დროში, ელემენტის ჩამატებას და არსებული ელემენტის შემცირებას ამორტიზებულ $O(1)$ დროში, ხოლო მინიმალური ელემენტის ამოშლას ამორტიზებულ $O(\log_2 N)$ დროში ახდენს.

თავი 2. არსებული ალგორითმები

ნაშრომში ძირითადი აქცენტი გაკეთებულია უმოკლესი გზის მოძებნის სამ მეთოდზე: დეიქსტრას ალგორითმზე, A* ძებნაზე და ორმხრივ დეიქსტრას ალგორითმზე.

შემდგომში ვიგულისხმოთ, რომ საძებნი გრაფის დასახელებაა G, ხოლო უმოკლესი გზების ძებნა s და t წვეროებს შორის მიმდინარეობს.

სამივე ალგორითმი იყენებს რელაქსაციის მექანიზმს (თუმცა A* ძებნისთვის იგი მცირეოდენ განსხვავებულია). ალგორითმის პროცესში, ყოველი v წვეროსთვის ინახება d[v] პარამეტრი - s წვეროდან v წვერომდე ნაპოვნი უმოკლესი გზის სიგრძე. d მასივის ინიციალიზაცია ალგორითმის დასაწყისში ხორციელდება შემდეგი პროცედურის მეშვეობით:

```
INITIALIZE-SINGLE-SOURCE(G, s)
  for each v in G.V
    d[v] = INFINITY
  d[s] = 0
```

(u, v) წიბოს რელაქსაცია (u -> v მიმართულებით) მდგომარეობს შემდეგში. მოწმდება, u-ს გავლით v-მდე გზის სიგრძე ნაკლები თუ არის მიმდინარე უმოკლეს გზაზე v წვერომდე და ასეთ შემთხვევაში ხდება v-მდე უმოკლესი გზის სიგრძის მნიშვნელობის განახლება. პსევდოკოდის სახით ეს პროცედურა შემდეგნაირად გამოიყურება:

```
RELAX(u, v, w)
  if d[v] > d[u] + w(u, v) then
    d[v] = d[u] + w(u, v)
```

2.1. დეიქსტრას ალგორითმი.

კლასიკური დეიქსტრას ალგორითმი [5][6] პოულობს G=(V, E) ორიენტირებული გრაფისთვის უმოკლეს გზებს საწყისი s წვეროდან ყველა დანარჩენ წვერომდე. მისი სწორად

მუშაობისთვის აუცილებელია, რომ გრაფის წიბოების წონები არაუარყოფითი რიცხვები იყოს.

დეიქსტრას ალგორითმის მუშაობის დროს გამოიყენება S სიმრავლე, რომელიც V-ს ქვესიმრავლეს წარმოადგენს და შედგები ისეთი v წვეროებისგან, რომლებამდეც უმოკლესი გზის სიგრძე უკვე ნაპოვნია. ალგორითმი იტერაციულია და ყოველ იტერაციაზე ხდება ისეთი u წვეროს არჩევა V \ S სიმრავლიდან, რომლის d[u] მინიმალურია. შემდეგ ხდება u წვეროს ყველა მეზობელი წიბოს რელაქსაცია.

პროცესი გრძელდება მანამ, სანამ V \ S სიმრავლეში არ რჩება მხოლოდ d[v]=INFINITY-ს მქონე წვეროები (რომლებიც მიუღწეველია s-იდან) ან რომელიმე იტერაციაზე არ ხდება t წვეროს არჩევა როგორც V \ S სიმრავლის მინიმალური d-ს მქონე წვეროსი.

```
DIJKSTRA(G, w, s, t)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S = {}
  while V \ S != {}
    u = extract_min(V \ S)
    S = S join {u}
    if u = t then
      break
    for each v adjacent to u
      RELAX(u, v, w)
  return d[t]
```

ალგორითმის სირთულე დამოკიდებულია გრაფის წარმოდგენის შერჩევასა და მონაცემთა სტრუქტურაზე, რომლის მეშვეობითაც ხორციელდება მანძილების შენახვა-განახლების ორგანიზება. მკვრივი გრაფებისთვის ოპტიმალურია მონაცემთა სტრუქტურის როლში მასივის აღება, რა შემთხვევაშიც ალგორითმის მუშაობის სირთულე იქნება $O(|V|^2)$. ხალვათი გრაფებისთვის გრაფი აუცილებლად მეზობელთა სიების სახით უნდა იყოს წარმოდგენილი და მონაცემთა სტრუქტურის როლში უნდა იქნას გამოყენებული ორობითი ან ფიბონაჩის

გროვა. ორობითი გროვის გამოყენებით სირთულე $O(|E|\log_2|V|)$ იქნება, ხოლო ფიბონაჩის გროვის საშუალებით - $O(|V|\log_2|V| + |E|)$.

2.2. A* ალგორითმი ევკლიდური გრაფებისთვის.

A* ძებნა [7] გამოიყენება იმ შემთხვევებში, როდესაც რაიმე ევრისტიული მეთოდის საშუალებით შესაძლებელია ყოველი წვეროდან დანიშნულების წვერომდე მანძილზე ქვედა ზღვრის შეფასება. ამ ცოდნის გათვალისწინებით, წვეროების განხილვა უფრო ეფექტური მიმდევრობით ხორციელდება, ვიდრე დეიქსტრას ალგორითმში და შესაბამისად ალგორითმი უფრო მალე ამთავრებს მუშაობას.

ევკლიდურ გრაფში ნებისმიერი წვეროდან დანიშნულების წვერომდე უმოკლეს გზაზე ქვედა ზღვრის შეფასება ბუნებრივია: ამ გზის სიგრძე ვერ იქნება შესაბამის წვეროებს შორის მანძილზე უფრო მოკლე. ამრიგად, შესაძლებელია წვეროები განხილული იქნას არა თუ ძებნის სათავიდან მანძილის ზრდის მიმდევრობით, არამედ ამ მანძილისა და დანიშნულების წვერომდე მანძილის ჯამის ზრდის მიმდევრობით. ეს მეთოდი ევრისტიულია და კონკრეტული სპეციალურად შერჩეული გრაფებისთვის შეიძლება ვერ აჯობოს დეიქსტრას ალგორითმს, მაგრამ მისი საშუალო შეფასება ასიმპტოტურად უკეთესია დეიქსტრას ალგორითმის შეფასებაზე.

ალგორითმის აღსაწერად გამოვიყენოთ მოდიფიცირებული INITIALIZE_SINGLE_SOURCE და RELAX პროცედურები:

```
INITIALIZE-SINGLE-SOURCE(G, s, t)
  for each v in G.V
    d[v] = INFINITY
  d[s] = distance(s, t)

RELAX(G, u, v, t)
  new_length = d[u] - distance(u, t) + w(u, v) + distance(v, t)
  if d[v] > new_length then
    d[v] = new_length
```

ამის შემდეგ, A* ძებნის იმპლემენტაცია ევკლიდური გრაფებისთვის პრაქტიკულად არ განსხვავდება დეიქსტრას ალგორითმის იმპლემენტაციისგან:

```
A*(G, w, s, t)
  INITIALIZE-SINGLE-SOURCE(G, s, t)
  S = {}
  while V \ S != {}
    u = extract_min(V \ S)
    S = S join {u}
    if u = t then
      break
    for each v adjacent to u
      RELAX(G, u, v, t)
  return d[t]
```

2.3. ორმხრივი დეიქსტრას ალგორითმი.

ორმხრივი დეიქსტრას ალგორითმის [8] იდეა შემდეგში მდგომარეობს: ერთდროულად მოხდეს დეიქსტრას ალგორითმის გაშვება s და t წვეროებიდან და პროცესი შეჩერდეს, როგორც კი ეს ძებნები ერთმანეთს „შეხვდებიან“. იმ შემთხვევაში, თუ საქმე ორიენტირებულ გრაფთან გვაქვს, t წვეროდან გაშვებული დეიქსტრა შებრუნებული გრაფის წიბოებით უნდა მოძრაობდეს. ერთდროულობის სიმულაცია ორი გზით შეიძლება: ყოველ ჯერზე ავირჩიოთ ისეთი წვერო, რომლამდეც მანძილი შესაბამისი გროვაში უფრო მცირეა, ანაც მონაცვლეობით ავიღოთ წვეროები ჯერ ერთი, შემდეგ კი მეორე გროვიდან. დაწვრილებით განვიხილოთ რას წარმოადგენს ალგორითმების „შეხვედრა“.

აღვნიშნოთ პირველი ალგორითმის მიერ გარკვეულ იტერაციაზე ნაპოვნი უმოკლესი მანძილები d1[·] სიდიდეებით, ხოლო მეორე ალგორითმის მიერ ნაპოვნი უმოკლესი მანძილები კი d2[·] სიდიდეებით. როდესაც პირველი ალგორითმი ახდენს რაიმე (u, v) წიბოს რელაქსაციას, ასევე მოწმდება d1[u] + w(u, v) + d2[v] სიდიდე და თუ იგი ნაკლებია ამ მომენტისთვის ნაპოვნ უმოკლეს L მანძილზე s და t წვეროს შორის (რომელიც თავიდან

პირობითად უსასრულობის ტოლია), მაშინ ხდება ამ მანძილის განახლება. ანალოგიური პროცედურა სრულდება მეორე ალგორითმის მიერ ჩატარებული რელაქსაციების დროსაც. ალგორითმი მთავრდება, როდესაც L ნაკლები ან ტოლი აღმოჩნდება d1[u']+d2[v'] სიდიდეზე, სადაც u' არის პირველი ალგორითმის მიერ დაუმუშავებელი s-თან უახლოესი წვერო, ხოლო v' არის მეორე ალგორითმის მიერ დაუმუშავებელი t-სთან უახლოესი წვერო. ამ დროისთვის L-ის მნიშვნელობა ოპტიმალურია.

ალგორითმი პსევდოკოდის სახით შემდეგნაირად გამოიყურება:

```
BIDIRECTIONAL-DIJKSTRA(G, w, s, t)
  INITIALIZE-SINGLE-SOURCE:d1(G, s)
  INITIALIZE-SINGLE-SOURCE:d2(rev(G), t)
  S1 = {}
  S2 = {}
  L = INFINITY
  while (V \ S1 != {}) AND (V \ S2 != {})
    u1 = extract_min(V1 \ S)
    u2 = extract_min(V2 \ S)
    if d[u1] + d[u2] >= INFINITY then
      break
    if d1[u1] < d2[u2] then
      S1 = S1 join {u1}
      for each v adjacent to u1 in G
        RELAX:d1(u1, v, w)
        L = min(L, d1[u1] + w(u1, v) + d2[v])
    else
      S2 = S2 join {u2}
      for each v adjacent to u2 in rev(G)
        RELAX:d2(u2, v, w)
        L = min(L, d2[u2] + w(u2, v) + d1[v])
  return L
```

დავასაბუთოთ ალგორითმის სისწორე. ამისთვის საკმარისია ვაჩვენოთ, რომ ალგორითმის დასასრულის მომენტში L-ში აუცილებლად იქნება უმოკლესი გზის სიგრძე s და t წვეროებს შორის. დავუშვათ საწინააღმდეგო: ვთქვათ, არსებობს ისეთი P გზა s-იდან t-ში, რომლის სიგრძე ნაკლებია L-ზე. ვიპოვოთ ამ გზის ბოლო წვერო u ისეთი, რომ d1[u] < d1[u'] დაპირველი წვერო v ისეთი, რომ d2[v] < d2[v']. თუ u წვერო ემთხვევა v-ს ან P-ში უფრო გვიან გვხვდება, ვიდრე v, მაშინ ალგორითმს P გზა განხილული ექნებოდა. თუ u და v P-ში მეზობელი წვეროებია, მაშინ P-ს განხილვა მოხდა იმ მომენტში, როდესაც u პირველმა დეიქსტრას ალგორითმმა დაამუშავა, ანაც იმ მომენტში როდესაც v მეორე დეიქსტრას ალგორითმმა დაამუშავა - რომელიც უფრო გვიან მოხდა. ვთქვათ, u და v-ს შორის 2 წიბო მაინც არის. პირველის წონა იყოს w1, ხოლო მეორის w2. იქიდან გამომდინარე, რომ L მეტია P-ს სიგრძეზე, გვაქვს d1[u]+w1+w2+d2[v] < L. ალგორითმის გაჩერების პირობიდან გამომდინარე გვაქვს L <= d1[u']+d2[v']. ხოლო u და v-ს განმარტებიდან გამომდინარე, გვაქვს d1[u]+w1 >= d1[u'] და d2[v]+w2 >= d2[v']. მივიღეთ წინააღმდეგობა.

კვლავ, ამ ალგორითმის სირთულე ზოგადად იგივეა რაც დეიქსტრას ალგორითმის სირთულე. თუმცა სპეციფიურ შემთხვევებში იგი თეორიულადაც ჯობია ჩვეულებრივ დეიქსტრას ალგორითმს. მაგალითად, თუ ჩავთვლით რომ გრაფი წარმოადგენს უწონო ხეს, რომლის ყველა წვეროს გააჩნია K+1 რაოდენობის მეზობელი, ხოლო s და t-ს ზუსტად K მეზობელი, ხოლო უმოკლესი გზა s-ს და t-ს შორის შედგება L წიბოსგან, მაშინ დეიქსტრას ალგორითმს მოუწევს K^L წვეროს განხილვა, ხოლო ორმხრივ დეიქსტრას ალგორითმს მხოლოდ $K^{L/2}$.

თავი 3. MinimumAngleDFS ევრისტიკა

MinimumAngleDFS წარმოადგენს ევრისტიულ ალგორითმს, რომელიც G ევკლიდურ გრაფში s წვეროდან t წვეროში მოსალოდნელად მოკლე (არა აუცილებლად უმოკლეს) გზას ეძებს. მისი იდეა იმაში მდგომარეობს, რომ ძებნისას პირველ რიგში მოსინჯოს ისეთი წიბოები, რომლებიც მიმდინარე v წვეროსგან t-სკენ უთითებენ ან v და t წვეროებით შექმნილ მონაკვეთთან მინიმალურ კუთხეს ქმნიან. ლოგიკურია, რომ საკმარისად მკვრივ გრაფში ასეთი სტრატეგიით t წვერო მალევე უნდა ვიპოვოთ და მიღებული გზის სიგრძე ხშირად შედარებადი იქნება უმოკლეს მანძილთან.

ალგორითმის პსევდოკოდი შემდეგია:

```
MINIMUM-ANGLE-DFS(G, s, t, used)
  if s = t then
    return 0
  used[s] = true
  sort vertices v adjacent to s in descending order of angle svt
  for each v adjacent to s
    if not used[v]
      path_length = MINIMUM-ANGLE_DFS(G, v, t, used)
      if path < INFINITY then
        return path + distance(s, v)
  return INFINITY
```

მკვრივ გრაფში წარმადობის თვალსაზრისით შეიძლება მომგებიანი აღმოჩნდეს წვეროების დალაგების ნაცვლად ერთი ან რამდენიმე მაქსიმალური კუთხის შემქმნელი წვეროს პოვნა და სიღრმეში ძებნის მათთვის გაშვება, რაც დიდი ალბათობით წარმატებით დასრულდება. ამით თავიდან ავიცილებთ მეზობლების დალაგების შედარებით შრომატევად პროცედურას.

თავი 4. იმპლემენტაცია

მიმდინარე თავში ხდება ყველა იმ გადაწყვეტილების მიმოხილვა, რომელიც იყო მიღებული იმპლემენტაციის ამა თუ იმ ეტაპზე, მათ შორის შემთხვევითი რიცხვების გენერატორის შერჩევა, შემთხვევითი გრაფების გენერაციის ოპტიმიზაცია, გრაფების, ალგორითმების და მათთვის საჭირო მონაცემთა სტრუქტურების განზოგადება.

პროექტი შესრულებულია C++ დაპროგრამების ენაზე.

4.1. შემთხვევითი რიცხვების გენერატორი.

ვინაიდან ალგორითმების სწრაფქმედებას სტატისტიკურად ვამოწმებდი, საჭირო იყო დიდი რაოდენობის შემთხვევითი გრაფის შექმნა და შესაბამისად უამრავი შემთხვევითი სიდიდის გენერაცია. ამრიგად, შემთხვევითი რიცხვების გენერატორი ერთის მხვრივ მაქსიმალურად სწრაფი უნდა ყოფილიყო, მეორეს მხვრივ კი საკმარისად დიდი პერიოდი ჰქონოდა.

არჩევანი შევაჩერე ჯორჯ მარსალიას ე.წ. Xorshift გენერატორზე [9]. ეს შემთხვევითი რიცხვების გენერატორი მხოლოდ ბიტურ ოპერაციებს (ბიტურ წაძვრას და xor-ს) იყენებს და შესაბამისად დიდი წარმადობით გამოირჩევა. C++-ის სტანდარტულ rand() ფუნქციასთან შედარებით Xorshift-ის მეშვეობით 10-მაგი მოგება მივიღე წარმადობაში. კონკრეტულად ვიყენებ შემდეგ იმპლემენტაციას, რომლის პერიოდია $2^{128}-1$:

```
uint32_t xor128(void) {
  static uint32_t x = 123456789;
  static uint32_t y = 362436069;
  static uint32_t z = 521288629;
  static uint32_t w = 88675123;
  uint32_t t;

  t = x ^ (x << 11);
  x = y; y = z; z = w;
  return w = w ^ (w >> 19) ^ (t ^ (t >> 8));
}
```

4.2. გრაფების წარმოდგენა.

გრაფების წარმოსადგენად ორი ძირითადი საშუალება არსებობს: მეზობლობის მატრიცის (Adjacency Matrix) და მეზობელთა სიების (Adjacency Lists) შენახვა. მეხსიერების დაკავების მხვრივ, პირველი ხერხი ოპტიმალურია მკვრივი, ხოლო მეორე ხალვათი გრაფებისთვის. ვინაიდან ალგორითმების შედარებისას ორივე სახის გრაფები იყო გამოყენებული, ეს ორივე წარმოდგენა იმპლემენტირებული უნდა ყოფილიყო მაქსიმალური სწრაფქმედებისთვის. ამრიგად, შეიქმნა EuclideanGraph აბსტრაქტული კლასი, რომელსაც აქვს შემდეგი ფუნქციონალი:

- void addEdge(int u, int v) - გრაფში (u, v) წიბოს დამატება
- int countNeighbours(int v) – v წვეროს მეზობლების რაოდენობის დათვლა
- int getFirstNeighbour(int v) – v წვეროს „პირველი“ მეზობლის დაბრუნება. პირველი პირობითი დასახელებაა და ფუნქციის შემდგომი იმპლემენტაციის მიხედვით შეიძლება აბრუნებდეს მინიმალური ნომრის მქონე მეზობელს ან იმ წვეროს, რომელიც v-სთან ყველაზე ადრე იყო დაკავშირებული წიბოების დამატების პროცესში
- int getNextNeighbour(int v) - აბრუნებს v წვეროს მორიგ მეზობელს. იგულისხმება, რომ ამ წვეროსთვის უკვე იყო გამოძახებული getFirstNeighbour მეთოდი. ეს ორი მეთოდი ერთად იძლევა წვეროს მეზობლების გადავლის საშუალებას.

EuclideanGraph-ს ორი იმპლემენტაცია აქვს DenseGraph და SparseGraph კლასების სახით.

DenseGraph ინახავს გრაფს მეზობლობის მატრიცის სახით. countNeighbours მეთოდის O(1) ალგორითმული სირთულის მისაღწევად ასევე ინახება ყოველი წვეროს მეზობელთა რაოდენობა. getFirstNeighbour და getNextNeighbour მეთოდების საშუალებით კონკრეტული წვეროს ყველა მეზობლის გადავლა O(N) დროში სრულდება, სადაც N გრაფში წვეროების რაოდენობაა.

SparseGraph ინახავს გრაფს მეზობელთა სიების სახით. getFirstNeighbour და getNextNeighbour მეთოდები კონკრეტული წვეროს მეზობლებს O(მეზობლების რაოდენობა) დროში გაივლიან, ანუ ჯამში ყველა წვეროსთვის O(M) დრო იხარჯება, სადაც M გრაფში წიბოების რაოდენობაა.

გრაფების ასე წარმოდგენა არ გვაძლევს საშუალებას, რომ საჭიროების შემთხვევაში (მაგალითად როდესაც გრაფში წიბოების დამატებისას იგი ხალვათის ნაცვლად მკვრივი ხდება) გრაფის სტრუქტურა შევცვალოთ და იგი მეზობელთა სიის ნაცვლად მატრიცით შევინახოთ. ამრიგად, გადაწყვეტილება შესაბამისი შენახვის გზის თაობაზე გრაფის შექმნისას უნდა იყოს მიღებული მომხმარებლის მიერ.

4.3. ბერნულის შემთხვევითი გრაფების გენერაცია.

გავიხსენოთ, რომ ბერნულის შემთხვევით G(n, p) გრაფში n წვეროა და წვეროების ყოველ წყვილს შორს წიბო p ალბათობით არის გავლებული.

პირველი საკითხი, რომლის გადაწყვეტაა საჭირო ასეთი გრაფის გენერაციის დროს არის გრაფის წარმოდგენის შერჩევა - შევინახოთ იგი მეზობლობის მატრიცით თუ მეზობელთა სიებით. ამისთვის საჭიროა საბოლოო გრაფში წიბოების რაოდენობის შეფასება.

ყოველი წიბოს არსებობის ალბათობა არის p, სულ კი არამიმართულ გრაფში $\frac{n\cdot(n-1)}{2}$ შესაძლო წიბოა. ასეთი მოდელი ბინომიალური განაწილებით აღიწერება. ამრიგად, ჩვენს გრაფში წიბოების რაოდენობის მათემატიკური მოლოდინი იქნება $\frac{n\cdot(n-1)}{2}p$. ასევე მნიშვნელოვანია, რა ალბათობით შეიძლება მოხდეს ამ რიცხვისგან არსებითი გადახრა. ჩავატარე ცდები კონკრეტულ მაგალითებზე შემდეგი ფუნქციის მეშვეობით:

```
double C[maxM][maxM];
double countBinDistDensity(int tries, double p, int left, int right)
{
	C[0][0] = 1.;
	for(int i = 1; i <= tries; i++)
	{
		C[i][0] = C[i - 1][0] * (1 - p);
		for(int j = 1; j <= i; j++)
			C[i][j] = C[i - 1][j - 1] * p + C[i - 1][j] * (1 - p);
	}

	double ret = 0.0;
	for(int i = max(0, left); i <= min(tries, right); i++)
		ret += C[tries][i];

	return ret;
}
```

ფუნქცია ითვლის ალბათობას, რომ tries რაოდენობის ცდის შედეგად, ყოველი რომელთაგანის წარმატების ალბათობაა p, წარმატებული ცდების რაოდენობა იქნება [left, right] შუალედში.

ცდებმა აჩვენა, რომ მაგალითად G(50, 0.5) ბერნულის შემთხვევითი გრაფისთვის იმის ალბათობა, რომ გრაფში წიბოების რაოდენობა 10%-ით განსხვავდება $\frac{n\cdot(n-1)}{2}p \approx 612$ -ისგან არის 0.1%-ზე ნაკლები. კიდევ უფრო ნაკლებია ალბათობა, რომ G(50, 0.01) გრაფში იქნება 30 წიბოზე მეტი (წიბოების რაოდენობის მათ. მოლოდინი ამ შემთხვევაში უდრის 12.25-ს). ეს ალბათობები კიდევ უფრო მცირდება n-ის გაზრდასთან ერთად. ამრიგად, მაღალი გადახრის ალბათობა საკმარისად დიდ გრაფში ძალიან მცირეა და ჩვენი მიზნებისთვის საბოლოო გრაფში წიბოების რაოდენობის მიახლოება ამ მათ.მოლოდინის მეშვეობით შეიძლება.

წიბოების მოსალოდნელ რაოდენობაზე დაყრდნობით გრაფის წარმოდგენის შერჩევის შემდეგ დგება გრაფის წიბოების გენერაციის საკითხი. გენერაციის მარტივი ალგორითმი შემდეგია: განვიხილოთ ყველა შესაძლო წყვილი, ყოველისთვის ავიღოთ [0, 1) შუალედში მყოფი შემთხვევითი რიცხვი და თუ იგი p-ზე ნაკლებია, დავამატოთ შესაბამისი წიბო გრაფში. ასეთი ალგორითმი მუშაობს $O(n^2)$ დროში და ოპტიმალურია მკვრივი გრაფებისთვის. მაგრამ n-ის დიდი მნიშვნელობებისთვის ყოველი გრაფის შექმნას საგრძნობი დრო დაეთმობა და ეს დრო თითქმის ერთიდაიგივე იქნება მკვრივი და ხალვათი გრაფებისთვის. ამიტომ ხალვათი გრაფებისთვის მივმართავ გენერაციის სხვა ალგორითმს.

პირველ რიგში, ვითვლი გრაფში წიბოების რაოდენობის მათ. მოლოდინს M. იმ შემთხვევაში, თუ გრაფი საკმარისად ხალვათი გამოდის, ასევე ვახდენ ბინომიალური განაწილების მოდელირებას $\left\lfloor\frac{M}{10}\right\rfloor$ ცდით და M-ის მნიშვნელობას ვაზუსტებ ამ ცდების შედეგის მიხედვით.

შემდეგ ხდება M რაოდენობის განსხვავებული (u, v) წყვილის გენერაცია და ამ წყვილების ჩამატება გრაფში წიბოებად. წყვილების უნიკალურობის შესამოწმებლად გამოიყენება ჰეშ-ცხრილი, რომელიც ელემენტის ჩამატებას და მოძებნას მოსალონდელ O(1) დროში ახდენს. კონკრეტულად გამოყენებულია C++-ის unordered_set კონტეინერი. შევაფასოთ ამ ალგორითმის სირთულე:

შემოვიღოთ აღნიშვნა $e = \frac{n\cdot(n-1)}{2}$. როდესაც ხდება რიგით მე-(k+1) წიბოს გენერაცია, იგი $\frac{k}{e}$ ალბათობით შეიძლება დაემთხვას რომელიმე იქამდე დაგენერირებულ წიბოს. მისი ხელმეორედ გენერაციისას ამავე ალბათობით შეიძლება კვლავ მოხდეს დამთხვევა და ასე შემდეგ. ამრიგად, მე-(k+1) წიბოს ჩამატების დროის მათ.მოლოდინი მოიცემა განტოლებით $T(k+1) = \frac{k}{e}\cdot T(k+1) + 1$. აქედან ვიღებთ $T(k+1) = \frac{e}{e-k}$.

როდესაც M=e, ანუ გრაფში ყველა წიბოს ჩამატება ხდება, T(1)+T(2)+...+T(M) ჯამი უდრის $\sum_{i=1}^{e}\frac{e}{i}$ -ს, რაც წარმოადგენს მე-e ჰარმონიულ რიცხვს და იზრდება O(e ln(e)) ტემპით [10]. ამრიგად, ამ ალგორითმის მუშაობის მოსალოდნელი დრო შემოსაზღვრულია O(e ln(e)) სირთულით და პრაქტიკაში უფრო დაბალია, ვინაიდან მხოლოდ ხალვათი გრაფებისთვის გამოიყენება.

ექსპერიმენტების შედეგად, შევარჩიე C=40 კონსტანტა და გენერაციის მარტივ ალგორითმს ვიყენებ მაშინ, თუ C * M აღემატება e-ს.

4.4. შემთხვევითი გეომეტრიული გრაფების გენერაცია.

გავიხსენოთ, რომ შემთხვევით გეომეტრიულ G(n, r) გრაფში წვეროების (u, v) წყვილი დაკავშირებულია მაშინ და მხოლოდ მაშინ, თუ ევკლიდური მანძილი u-ს და v-ს შესაბამის წერტილებს შორის არ აღემატება r-ს.

ბერნულის შემთხვევითი გრაფების გენერაციის მსგავსად, აქაც პირველი გადასაწყვეტი საკითხია გრაფის წარმოდგენის შერჩევა. საამისოდ კვლავ შევაფასოთ წიბოების მოსალოდნელი რაოდენობა, რომელიც იქნება G(n, r) გრაფში. მათემატიკური მოლოდინის წრფივობიდან გამომდინარე, საკმარისია გამოვთვალოთ ალბათობა p, რომ d-განზომილებიანი სივრციდან შემთხვევით აღებული 2 წერტილი იქნება არაუმეტეს r მანძილით დაშორებული, შემდეგ კი გავამრავლოთ ეს p სიდიდე პოტენციური წიბოების $\frac{n\cdot(n-1)}{2}$ რაოდენობაზე.

d=2 შემთხვევაში, ანუ სიბრტყეზე განლაგებული წერტილებისთვის, ამოცანას ანალიზურად შეგვიძლია მივუდგეთ. დავადგინოთ p(r) დამოკიდებულება: ალბათობა, რომ ერთეულოვანი

კვადრატიდან აღებული ორი შემთხვევითი წერტილი არაუმეტეს r მანძილით არის დაშორებული. მოხერხებულობისთვის, გამოვთვლი ამის საპირისპიროს - რომ ორი წერტილი r-ზე მეტი მანძილითაა დაშორებული. აქ განვიხილავ მხოლოდ r≤1 შემთხვევას, რაც გაამარტივებს ანალიზის ჩატარებას და ასევე არ წარმოადგენს სერიოზულ შეზღუდვას: r>1 შემთხვევაში საკმარისად დიდი გრაფი ყოველთვის მკვრივი გამოდის.

ვთქვათ, ჩვენი წერტილებია A და B. განვიხილოთ U=AB=(x, y) ვექტორი. ზოგადობის შეუზღუდავად ჩავთვალოთ, რომ კუთხე, რომელსაც U ვექტორი ქმნის OX ღერძთან, $[0, \frac{\pi}{4})$ შუალედში მდებარეობს (სიმეტრიულობიდან გამომდინარე, სხვა სექტორებისთვის ზუსტად იგივე ალბათობა მიიღება). ამრიგად, მიღებული ალბათობა 8-ზე უნდა გავამრავლოთ საბოლოო პასუხის მისაღებად.

ფიქსირებული U=(x, y) ვექტორისთვის A შეიძლება იყოს ნებისმიერი წერტილი [1-x]x[1-y] მართკუთხედიდან (ვინაიდან B=A+U ერთეულოვანი კვადრატის შიგნით უნდა მოხვდეს). ამრიგად, ასეთი U ვექტორის „წონა“ არის (1-x)(1-y). გამოვთვალოთ ამ გამოსახულების ინტეგრალი ყველა დაშვებულ U ვექტორზე:

$$\iint\limits_{0<x<y<1}^{x^2+y^2>r^2} (1-x)(1-y)dxdy$$

ამის გამოსათვლელად გადავიდეთ პოლარულ კოორდინატებში $\iint f(x,y)dxdy = \iint Rf(Rsin\theta, Rcos\theta)dRd\theta$ ფორმულის გამოყენებით:

$$\iint\limits_{\theta=0,R=r}^{\frac{\pi}{4},\frac{1}{cos\theta}} R(1-Rcos\theta)(1-Rsin\theta)dRd\theta$$

ინტეგრალის ქვეშ მყოფ გამოსახულებაში ფრჩხილების გახსნით მივიღებთ $R^3 sin\theta cos\theta - R^2(sin\theta + cos\theta) + R$, ამ გამოსახულების გაინტეგრებით (შიდა ინტეგრალით) კი $\left(\frac{R^4}{4} sin\theta cos\theta - \frac{R^3}{3}(sin\theta + cos\theta) + \frac{R^2}{2}\right)\Big|_r^{1/cos\theta} = \frac{2cos\theta - sin\theta}{12cos^3\theta} - \frac{3r^4 sin\theta cos\theta - 4r^3(sin\theta + cos\theta) - 6r^2}{12}$. ახლა ავიღოთ ამ გამოსახულების ინტეგრალი θ-ს მიმართ:

$$\int_0^{\frac{\pi}{4}} \frac{2cos\theta - sin\theta}{12cos^3\theta} - \frac{3r^4 sin\theta cos\theta - 4r^3(sin\theta + cos\theta) - 6r^2}{12} d\theta =$$

$$\frac{r^4 cos^2\theta}{8} + \frac{r^3(sin\theta - cos\theta)}{3} - \frac{r^2\theta}{2} + \frac{tan\theta}{6} - \frac{1}{24cos^2\theta}\Big|_0^{\pi/4} = -\frac{r^4}{16} + \frac{r^3}{3} - \frac{\pi r^2}{8} + \frac{1}{8}$$

იმის გათვალისწინებით, რომ მიღებული შედეგი არის სიბრტყის 1/8 ნაწილისთვის და ასევე რომ ეს იმის შებრუნებული ხდომილების ალბათობაა, რასაც ვეძებდით, ვიღებთ p(r) დამოკიდებულების საბოლოო ფორმულას r≤1 შემთხვევისთვის:

$$p(r) = \frac{r^4}{2} - \frac{8}{3}r^3 + \pi r^2$$

ვინაიდან მაღალი განზომილებებისთვის ანალიზის ჩატარება უფრო რთულ ამოცანას წარმოადგენს, d>2 შემთხვევებში p(r) სიდიდის შესაფასებლად მონტე-კარლოს მეთოდს ვიყენებ: ვატარებ დიდი რაოდენობის (10 მილიონ) ცდას და ვითვლი ორი შემთხვევითი წერტილის არაუმეტეს r მანძილზე აღმოჩენის სიხშირეს. შედეგად მიღებული ალბათობები მაქსიმუმ მიახლოებით 0.00025-ით განსხვავდებოდა რეალური სიდიდეებისგან d=2 შემთხვევაში.

უშუალოდ გრაფის წიბოების გენერაციისთვის კვლავ შეიძლება მარტივი ალგორითმის გამოყენება: წვეროების ყოველი (u, v) წყვილის განხილვა, მათ შორის მანძილის გამოთვლა და გრაფში (u, v) წიბოს ჩამატება, თუ ეს მანძილი არ აღემატება r-ს. მეტიც, აქ შეიძლება საკმაოდ მძიმე ფესვის ამოღების ფუნქციისგან თავის დაღწევა, თუ გამოვთვლით მანძილის კვადრატს და შევადარებთ r^2 სიდიდეს. ამრიგად, კვლავ გვაქვს $O(n^2)$ სირთულის ალგორითმი.

წერტილების შემთხვევითი განაწილებისთვის და დიდი r-ისთვის ეს ალგორითმი ოპტიმალურთან ახლოსაა. როდესაც წერტილების განაწილება არ არის შემთხვევითი, ანაც r-ის მნიშვნელობა საკმარისად დაბალია, არსებობს უკეთესი მეთოდები. საზოგადოდ, წერტილისგან არაუმეტეს R მანძილით დაშორებული წერტილების მოძებნის ამოცანას ეძახიან „Fixed-radius near neighbours“ და იგი მრავალი კუთხით არის გამოკვლეული [11]. ჩემს შემთხვევაში გამოვიყენე ერთ-ერთი ცნობილი Cell Technique მეთოდის მოდიფიკაცია.

ჩემი მიდგომა d=2 შემთხვევაში შემდეგში მდგომარეობს. ავიღოთ C = $\left\lfloor \frac{1}{r} \right\rfloor$ რიცხვი და დავყოთ ერთეულოვანი კვადრატი C x C მატრიცად. მატრიცის ყოველი უჯრისთვის შემოვიღოთ იმ წვეროების სია, რომლებიც მის შიგნით მოხვდა. ვინაიდან ყოველი უჯრის გვერდის სიგრძე არის r ან მეტი, ესე იგი ნებისმიერი წერტილისგან არაუმეტეს r მანძილით დაშორებული წერტილები იქნება ან იმავე უჯრაში რაც ეს წერტილი, ან ერთ-ერთში 8 მეზობელი უჯრისგან. ამრიგად, საშუალოდ ხდება მხოლოდ $\frac{9}{C\hat{}2}n \approx 9nr\hat{}2$ წერტილის შემოწმება ყოველი წერტილისთვის.

4.5. მონაცემთა სტრუქტურები უმოკლესი გზის ძებნის ალგორითმებისთვის.

დეიქსტრას ალგორითმი და მასზე დაფუძნებული მეთოდები მკვრივი და ხალვათი გრაფების შემთხვევაში სხვადასხვა მონაცემთა სტრუქტურას იყენებენ. მკვრივი გრაფის შემთხვევაში $O(n^2)$ ალგორითმული სირთულის მისაღებად საკმარისია მიღწეულ წვეროებამდე უმოკლესი მანძილები მასივში იქნას შენახული, ხოლო ხალვათი გრაფისთვის $O(m \log_2 n)$ ან $O(n\log_2 n + m)$ სირთულე მიიღწევა გროვების და ფიბონაჩის გროვების საშუალებით.

ამ ნაშრომში ვიყენებ მხოლოდ მასივზე და ორობით გროვაზე დაფუძნებულ იმპლემენტაციებს. ფიბონაჩის გროვებზე უარი ვთქვი იმ მოსაზრებიდან გამომდინარე, რომ ალგორითმების სტატისტიკური შედარებისთვის საჭიროა მათი სწრაფქმედების დიდი რაოდენობის გრაფისთვის შედარება და აქედან გამომდინარე ეს გრაფები ვერ იქნება ძალიან დიდი. ფიბონაჩის გროვა კი პრაქტიკაში ორობით გროვასთან შედარებით უკეთეს შედეგებს იძლევა დაწყებული მილიონწვეროიანი გრაფებიდან.

ნაშრომის ფარგლებში შექმნილია კლასი DataCenter და მისი შვილობილი კლასი HeapDataCenter. იგულისხმება, რომ ეს სტრუქტურები ინახავენ მთელ ინფორმაციას ალგორითმის მსვლელობის შესახებ, მათ შორის მანძილებს მიღწეულ წვეროებამდე და მიღწევადობის ჭდეებს.

ამ სტრუქტურებს გააჩნიათ შემდეგი მეთოდები:

- void decreaseKey(int v, int dist) – v წვერომდე უმოკლესი გზის სიგრძეს dist სიდიდემდე ამცირებს
- void markVisited(int v) - აღნიშნავს, რომ შესრულდა დეიქსტრას ალგორითმის იტერაცია v წვეროსთვის
- int getTopVertex() - აბრუნებს იმ წვეროს ნომერს, რომლისთვისაც ალგორითმის იტერაცია ჯერ არ შესრულებულა და რომელიც ყველა ასეთ წვეროს შორის ყველაზე ახლოს არის ძებნის სათავესთან, ან რიცხვს "-1" თუ ასეთი წვერო არ მოიძებნა

DataCenter ამ მეთოდების იმპლემენტაციას მასივის საშუალებით ახდენს. decreaseKey და markVisited მეთოდების ალგორითმული სირთულე O(1) აქვს, ხოლო getTopVertex კი O(n).

HeapDataCenter იყენებს მოდიფიცირებულ ორობით გროვას getTopVertex მეთოდის დასწრაფების მიზნით. გროვაში ჩამატებულია წვეროების ნომრები, რომლების შედარება ხდება შესაბამისი წვეროების მანძილებით და რომლებიც იცავენ გროვის ძირითად თვისებას. ასევე შემოღებულია „უკუინდექსაცია" (reverse index), რომლის საშუალებით შესაძლებელია მოცემული ნომრის მქონე წვეროს O(1) დროში მოძებნა გროვის სტრუქტურაში. ამრიგად, decreaseKey და markVisited მეთოდების სირთულე $O(\log_2 n)$ არის, ხოლო getTopVertex O(1)-ში სრულდება.

4.6. დეიქსტრას ალგორითმი და A* ძებნა.

დეიქსტრას ალგორითმის და A* ძებნის მსგავსებიდან გამომდინარე, ეს ორი ალგორითმი გაერთიანებული მაქვს ერთ მეთოდად, რომლის სიგნატურაა:

```
getShortestPath(EuclideanGraph & graph, int source, int destination, bool
heuristic, double upperBound = INFINITY, implementation imp = automatic)
```

განვმარტოთ პარამეტრების შინაარსი:

- graph - ევკლიდური გრაფი, რომელშიც მოხდება უმოკლესი გზის ძებნა
- source და destination - წვეროები, რომელთა შორის უმოკლესი გზის მოძებნაა საჭირო

- heuristic - ამ პარამეტრის მცდარი (false) მნიშვნელობის შემთხვევაში ხდება ჩვეულებრივი დეიქსტრას ალგორითმის ამუშავება. წინააღმდეგ შემთხვევაში ფუნქცია A* ძებნას ახორციელებს
- upperBound - ცნობილი ზედა ზღვარი უმოკლესი გზის სიგრძეზე
- imp - `enum implementation {automatic, matrix, heap};` ტიპის ცვლადი, რომელიც უთითებს, თუ რომელი მონაცემთა სტრუქტურა უნდა იქნას გამოყენებული ალგორითმის მსვლელობაში. automatic ნიშნავს, რომ ალგორითმი თვითონ შეარჩევს ხელსაყრელ მონაცემთა სტრუქტურას.

შეიძლება ლოგიკურად მოგეჩვენოთ heuristic პარამეტრის ნაცვლად ალგორითმისთვის წვეროების პოტენციალების მასივის გადაცემა. A* ძებნის შემთხვევაში ეს იქნებოდა მანძილები წვეროდან destination წვერომდე, ხოლო დეიქსტრას ალგორითმის შემთხვევაში - 0-ებით შევსებული მასივი (დეიქსტრას ალგორითმი შეიძლება A* ალგორითმის კერძო შემთხვევად მივიჩნიოთ ნულის ტოლი პოტენციალის ფუნქციით). მაგრამ ამ შემთხვევაში, A* ალგორითმისთვის საჭირო გახდებოდა ყველა წვეროსთვის destination წვერომდე მანძილის გამოთვლა, რაც sqrt ფუნქციის გათვალისწინებით საკმაოდ შრომატევად პროცესს წარმოადგენს.

upperBound პარამეტრის გამოყენება შემდეგში მდგომარეობს. თუ უმოკლესი მანძილის გამოთვლის ალგორითმამდე რაიმე ევრისტიული მეთოდით მოვახერხეთ რომელიმე გზის მოძებნა მოცემულ წვეროებს შორის, შეგვიძლია ეს ცოდნა გამოვიყენოთ ზედმეტი ოპერაციების თავიდან ასაცილებლად: თუ ვაპირებთ ისეთი წიბოს გამოყენებით რაიმე წვეროს გასაღების შემცირებას, რომლის მეორე ბოლოში მყოფი წვერო upperBound-ზე უფრო დიდ მანძილზე იმყოფება ძებნის სათავიდან, ეს წვერო ნამდვილად ვერ შევა საძებნ უმოკლეს გზაში და ამ გასაღების შემცირების ოპერაციის იგნორირება შეიძლება. გაჩუმებით ამ პარამეტრის მნიშვნელობა პირობით უსასრულობად არის ჩათვლილი, რაც ალგორითმის მსვლელობაზე ზეგავლენას არ ახდენს.

წიბოების სიგრძეები ალგორითმს ცხადად არ გადაეცემა. მათი გამოთვლა ხდება იმ დროს, როდესაც ამ წიბოების რელაქსაცია ხდება, შესაბამის წვეროებს შორის ევკლიდური მანძილის გამოთვლის მეშვეობით.

გრაფის სიმკვრივის ხარისხიდან გამომდინარე, ალგორითმი თვითონ ირჩევს მონაცემთა სტრუქტურას. შევნიშნოთ, რომ ეს არჩევანი არაა პირდაპირ დამოკიდებული იმაზე, თუ რა სახით არის წარმოდგენილი გრაფი: ზოგჯერ მეზობელთა სიებით შენახული გრაფისთვისაც უმჯობესია $O(n^2)$ სირთულის ალგორითმის ამუშავება. კონკრეტულად, HeapDataCenter სტრუქტურის არჩევა ხდება იმ შემთხვევაში, თუ $3*m*\log_2 m$ არ აღემატება n^2-ს (სადაც n და m გრაფში წვეროების და წიბოების რაოდენობებია შესაბამისად). ეს სიდიდეები ექსპერიმენტალურად იყო გამოყვანილი. შენიშვნა: როდესაც imp პარამეტრის მნიშვნელობა არ არის automatic, ალგორითმი იძულებულია შესაბამისი სტრუქტურა გამოიყენოს.

4.7. ორმხრივი დეიქსტრას ალგორითმი.

ორმხრივი დეიქსტრას ალგორითმი იმპლემენტირებულია getShortestPathDoubleSearch ფუნქციის სახით, რომლის სინგატურაა:

```
getShortestPathDoubleSearch(EuclideanGraph & graph, int source, int
destination, bool alternate)
```

graph წარმოადგენს გრაფს, რომელშიც ხდება უმოკლესი გზის ძიება, ხოლო source და destination იმ წვეროების წყვილია, რომელთა შორის უმოკლესი გზის მოძებნაა საჭირო.

დამატებითი პარამეტრი alternate-ის მნიშვნელობა შემდეგია: თუ იგი არის false, მაშინ ყოველ ნაბიჯზე ხდება იმ ნახევრიდან წვეროს დამუშავება, რომელიც თავის სათავესთან უფრო ახლოს იმყოფება. alternate=true შემთხვევაში წვეროების დამუშავება ყოველთვის მონაცვლეობით ხორციელდება: პირველი ნახევრიდან, მეორიდან, პირველიდან და ასე შემდეგ.

წინა ფუნქციის მსგავსად, აქაც ალგორითმი თვითონ წყვეტს, რომელი მონაცემთა სტრუქტურა გამოიყენოს ინფორმაციის დასამუშავებლად. HeapDataCenter სტრუქტურის არჩევა ხდება თუ $3mlog(m) \leq n^2$.

4.8. ალგორითმების სწრაფქმედების შედარება.

ვინაიდან ალგორითმების წარმადობის შედარება მრავალი სხვადასხვა სცენარისთვის იყო საჭირო, მიზანშეწონილი იყო მაქსიმალურად ზოგადი ფუნქციის შექმნა, რომელიც მოცემული პარამეტრების გრაფებისთვის მოახდენდა ალგორითმების გაშვებას და სტატისტიკის შეგროვებას. ასეთი ფუნქციის სიგნატურა შემდეგია:

```
void compareAlgorithms(int n, double p, int d,
EuclideanGraph* (*generator)(int,double,int), char * graphName,
char * filename, int tests, int queriesPerTest)
```

n, p, d წარმოადგენს გრაფის პარამეტრებს: წვეროების რაოდენობას, წიბოს გავლების ალბათობას ან წიბოს შესაქმნელად საჭირო მაქსიმალურ მანძილს, იმის მიხედვით თუ რა ტიპის გრაფთან იქნება საქმე და ევკლიდური სივრცის განზომილება.

generator წარმოადგენს შემთხვევითი გრაფის შემქმნელ ფუნქციაზე პოინტერს და შესაბამისად როგორც getBernoulliRandomGraph, ასევე getRandomGeometricGraph შეიძლება იყოს.

graphName და filename სტატისტიკის ფაილის პარამეტრებს წარმოადგენენ.

tests პარამეტრი განსაზღვრავს, თუ რამდენი სხვადასხვა გრაფი უნდა დაგენერირდეს ალგორითმების შესამოწმებლად. ყოველი ასეთი გრაფისთვის queriesPerTest რაოდენობის წვეროების შემთხვევითი წყვილი აირჩევა და ალგორითმების გაშვება ყოველი ამ წყვილისთვის ხდება.

ერთ-ერთი არსებითი მომენტია არაბმული გრაფების დამუშავება. იმ შემთხვევაში, როდესაც არაბმული გრაფის ორი სხვადასხვა კომპონენტის წვეროს ვირჩევთ უმოკლესი მანძილის დამდგენი რომელიმე ალგორითმის შემავალ მონაცემებად, ამ ალგორითმს უწევს მთელი შესაბამისი კომპონენტის შემოვლა, სანამ დაასრულებს მუშაობას. ეს გადაგვარებული შემთხვევა არსებითად ცვლის სტატისტიკურ სურათს, რის გამოც დაბალი p ან r პარამეტრის მნიშვნელობისთვის ვერ მივიღებდით რეალურ შედეგებს.

ასეთი სიტუაციის თავიდან ასაცილებლად გრაფის გენერაციის შემდეგ ვახდენ მის ბმულ კომპონენტებად დაშლას. როდესაც ხდება წვეროების წყვილების არჩევა, რომლებზეც

მოხდება ალგორითმების გაშვება, ხდება ყველა განსხვავებულ კომპონენტებში მყოფი წყვილის უგულებელყოფა.

4.9. MINIMUM-ANGLE-DFS მეთოდის იმპლემენტაცია.

ნაშრომის ფარგლებში იმპლემენტირებულია MINIMUM-ANGLE-DFS ალგორითმის ვერსია, რომელიც პირველ რიგში ეძებს საუკეთესო კანდიდატ მეზობელს და სიღრმეში ძებნას მისთვის იძახებს, ხოლო წარუმატებლობის შემთხვევაში ახდენს ყველა დანარჩენი მეზობლის დალაგებას და დალაგების თანმიმდევრობით აგრძელებს მათში ძიებას.

კიდევ ერთი გადასაწყვეტი ამოცანა ამ მეთოდის იმპლემენტაციაში არის კუთხეების გამოთვლა. გავიხსენოთ, რომ როდესაც ალგორითმი ამუშავებს რომელიმე v წვეროს, მისი მეზობელი წვეროები ლაგდება იმ კუთხის კლების მიხედვით, რომელიც იქმნება v წვეროს, შესაბამის მეზობელ u-ს და ძებნის დანიშნულების წერტილ t წვეროს შორის. როდესაც ეს კუთხე 180 გრადუსის ტოლია, ეს ნიშნავს რომ u წვერო მდებარეობს v წვეროდან t წვეროსკენ წამოსულ სხივზე და უფრო დიდი ალბათობით წარმოადგენს ძებნის გასაგრძელებლად ხელსაყრელ კანდიდატს, ვიდრე წვერო, რომელიც ქმნის 90-გრადუსიან კუთხეს (ანუ ამ სხვივისგან მართობული მიმართულებით მდებარეობს). ამ კუთხეების სიდიდეების ზუსტად გამოთვლა საკმაოდ შრომატევადი პროცესია, რომელიც მნიშვნელოვნად შეანელებდა ჩვენს პროცედურას. ამიტომ კუთხეების ნაცვლად ვახდენ მათი კოსინუსების გამოთვლას. კოსინუსების თეორემის თანახმად, საძებნი კუთხის კოსინუსი მოიცემა ფორმულით: $cos\gamma = \frac{a^2+b^2-c^2}{2ab}$, სადაც a = distance(v, t), b = distance(v, u), c = distance(u, t). ამ ფორმულის შეფასება შეიძლება მხოლოდ ერთი კვადრატული ფესვის ამოღების ოპერაციის საშუალებით, რაც გამოთვლას საკმაოდ ეფექტურს ხდის.

[0, 180]-გრადუსიან შუალედში კოსინუსები აკმაყოფილებენ მნიშვნელოვან თვისებას: კუთხის კოსინუსი კლებულობს თვითონ კუთხის ზრდასთან ერთად. ამრიგად, მეზობელი წვეროების დალაგება კუთხის კლებადობის მიხედვით ჩავანაცვლე მათი კოსინუსების ზრდის მიხედვით დალაგებით.

თავი 5. შედეგები

ალგორითმების წარმადობის შესაფასებლად მრავალი კრიტერიუმის მოფიქრება შეიძლება: მუშაობის რეალური დრო, ალგორითმის მიერ დამუშავებული წვეროების საშუალო რაოდენობა, ალგორითმის მიერ შესრულებული რელაქსაციების რაოდენობა, etc. ჩემს მიერ მოგროვებული იყო რეალური დროის გარდა ყველა ასეთი სახის სტატისტიკური მონაცემი. ძირითად პარამეტრად მათ შორის მიმაჩნია ალგორითმის მიერ დამუშავებული წვეროების რაოდენობის და ნაპოვნ უმოკლეს გზაში შესული წვეროების რაოდენობის საშუალო შეფარდება. სხვა სიტყვებით რომ ვთქვათ, ეს სტატისტიკა ითვლის, უმოკლესი გზის ყოველი წვეროს საპოვნელად საშუალოდ რამდენი წვეროს დამუშავება მოუწია ალგორითმს. ამ პარამეტრს შემდგომში ratio-თი მოვიხსენიებ.

ვინაიდან წინასწარი ტესტირების პერიოდში ორმხრივმა დეიქსტრას ალგორითმის იმპლემენტაციამ, რომელიც s-თან და t-სთან უახლოეს წვეროების მონაცვლეობით ამუშავებს, უკეთესი შედეგები აჩვენა, აქ ეს იმპლემენტაცია იყო გამოყენებული.

სიმულაციის სრული შედეგები დანართში არის მოცემული. აქ მოხდება ძირითადი შედეგების შეჯამება კლასიკური ალგორითმების წარმადობასა და MINIMUM-ANGLE-DFS მეთოდთან დაკავშირებით.

5.1. უმოკლესი გზის ალგორითმების შედარება ბერნულის შემთხვევითი გრაფებისთვის.

ბერნულის შემთხვევით გრაფებზე ალგორითმების შედარებისთვის შევარჩიე გრაფების სამი სხვადასხვა ზომა: წვეროების რაოდენობებით n=500, n=5000 და n=50000. ყოველი ზომის გრაფებისთვის ასევე შევარჩიე p-ს რამდენიმე მნიშვნელობა და ალგორითმების შედარება მოვახდინე დიდი რაოდენობის G(n, p) გრაფზე 2-, 3- და 4-განზომილებიან სივრცეში.

დეიქსტრას ალგორითმი სხვა ორთან შედარებით ყველაზე არაეკონომიურად მუშაობს. მის მიერ დამუშავებული წვეროების საშუალო რაოდენობა სტაბილურად რჩება გრაფის წვეროების რაოდენობის ნახევართან ახლოს. ამრიგად, სიბრტყეზე განლაგებულ 500-

წვეროიან ევკლიდურ გრაფებში p=0.01 პარამეტრის მნიშვნელობით, როდესაც გრაფში საშუალოდ 1186 წიბო გენერირდება და შემთხვევით აღებული წვეროების წყვილს საშუალოდ 5 წიბო აშორებს, მისი ratio მიახლოებით 50 გამოდის. ეს რიცხვი გრაფის ზომის, p-ს და განზომილების ზრდასთან ერთად კიდევ იზრდება.

გაცილებით უფრო საინტერესოა A* და ორმხრივი დეიქსტრას ალგორითმების მუშაობის სტატისტიკა. A* ძებნის წარმადობა შედარებით დაბალია ძალიან ხალვათი გრაფებისთვის (თუმცა მათთვისაც იგი საშუალოდ ბევრად ჯობნის დეიქსტრას ალგორითმს) და ასევე ცოტათი უარესდება სივრცის განზომილების ზრდასთან ერთად. ორმხრივი დეიქსტრა, პირიქით, მით უკეთესად მუშაობს, რაც უფრო ხალვათია გრაფი და რაც მეტია მისი განზომილება. მეტიც, ფიქსირებული d-სთვის ორივე ალგორითმის ratio მიახლოებით მონოტონურია p-ს მიმართ. აქედან გამომდინარე, სტატისტიკურად ყოველი n-სთვის არსებობს ისეთი p' ალბათობა, რომ G(n, p') გრაფზე A* და ორმხრივი დეიქსტრას წარმადობა მიახლოებით ერთნაირია, ხოლო p<p' მნიშვნელობებისთვის ორმხრივი დეიქსტრა საშუალოდ უკეთეს შედეგებს აჩვენებს და p>p' მნიშვნელობებისთვის შებრუნებული სურათი გვხვდება. შემოვიღოთ შესაბამისი p'(n) ფუნქცია.

დამატებითი ტესტირების შედეგად ვიპოვე p'(n) ფუნქციის მიახლოებითი მნიშვნელობები გრაფში წვეროების n=1000, 2000, 3000, ..., 9000 რაოდენობებისთვის d=2 სივრცის განზომილებისთვის:

n	~p'	m=p'*n*(n-1)/2	m/n
1000	0.0205	10239.75	10.23975
2000	0.0129	25787.1	12.89355
3000	0.0095	42735.75	14.24525
4000	0.0078	62384.4	15.5961
5000	0.0067	83733.25	16.74665
6000	0.0058	104382.6	17.3971
7000	0.0054	132281.1	18.8973
8000	0.0048	153580.8	19.1976
9000	0.0043	174130.65	19.34785

სამწუხაროდ, ამოცანის შინაარსიდან გამომდინარე ამ სიდიდეების ცდომილება ზედმეტად მაღალია p'(n) ფუნქციის მიახლოებითი სახის დასადგენად.

დასკვნა - ძალიან ხალვათი ბერნულის შემთხვევითი გრაფების შემთხვევაში ორმხრივი დეიქსტრა კლასიკური ალგორითმებს შორის საუკეთესო წარმადობით გამოირჩევა. ხალვათობის საჭირო კოეფიციენტი დამოკიდებულია გრაფში წვეროების რაოდენობასა და სივრცის განზომილებაზე. უფრო მკვრივი გრაფებისთვის უკეთესია A* ძებნის გამოყენება, რომელიც საშუალოდ გზების მოძებნას ძალიან სწრაფად ახერხებს.

5.2. უმოკლესი გზის ალგორითმების შედარება შემთხვევითი გეომეტრიული გრაფებისთვის.

შემთხვევითი გეომეტრიული გრაფებისთვის A* ალგორითმის უპირატესობა დეიქსტრას ორმხრივ ალგორითმთან შედარებით კიდევ უფრო გამოკვეთილი ხდება. ეს განპირობებულია შემთხვევითი გეომეტრიული გრაფების სტრუქტურით: რაც უფრო მცირეა r, მით უფრო დიდი სიგრძის იქნება შემთხვევითი უმოკლესი გზები და შესაბამისად მით უფრო დიდი რაოდენობის წვეროს მოინახულებს ორმხრივი დეიქსტრას ალგორითმი. მაღალი r-ებისთვის კი გრაფი ძალიან მკვრივი ხდება და ჩვენ უნდა დავადგინეთ ბერნულის გრაფების მაგალითზე, რომ ორმხრივი დეიქსტრას ალგორითმის წარმადობა ეცემა გრაფის წიბოების ზრდასთან ერთად.

ვინაიდან შემთხვევით გეომეტრიულ გრაფს უფრო დეტერმინისტული სტრუქტურა გააჩნია, ვიდრე ბერნულის შემთხვევით გრაფს, A* ალგორითმი აქ უკეთეს ratio-ს აღწევს. მაგალითად, სიბრტყეზე განლაგებულ 5000-წვეროიან და დაახლოებით 625,000-წიბოიან გრაფში მისი ratio 6-თან ახლოსაა, ხოლო ორჯერ მეტი წიბოს შემთხვევაში ratio 3.5-მდე ეცემა. საერთოდ, როდესაც გრაფში წიბოების რაოდენობა 3ჯერ მაინც აღემატება წვეროების რაოდენობას, ამ მეთოდის ratio 10-ზე უფრო დაბალია.

უფრო მაღალი განზომილების სივრცეებში A* ალგორითმის ეფექტურობა მკვეთრად ეცემა (თუმცა მაინც ორმხრივ დეიქსტრაზე ბევრად უკეთესი რჩება).

დასკვნა - შემთხვევით გეომეტრიულ გრაფებზე თითქმის ყოველთვის A* ალგორითმი უკეთესი წარმადობით გამოირჩევა, ვიდრე დეიქსტრას ალგორითმი და დეიქსტრას ორმხრივი ალგორითმი.

5.3. MINIMUM-ANGLE-DFS მეთოდის შეფასება.

MINIMUM-ANGLE-DFS ალგორითმი შემუშავებული იყო უმოკლესი სიგრძის ალგორითმებისთვის გზის სიგრძეზე ზედა შეფასების მისაწოდებლად. ამ შეფასების გამოყენებით, ისინი შეძლებდნენ ზედმეტი ოპერაციების თავიდან არიდებას იმ წვეროების იგნორირების ხარჯზე, რომლებიც ამ ცნობილ ზედა ზღვარზე უფრო შორს მდებარეობენ ძებნის სათავიდან.

როგორც აღმოჩნდა, ბერნულის შემთხვევით გრაფებში ეს მეთოდი ვერ აღწევს ბევრად უკეთეს წარმადობას, ვიდრე A* ძებნის ალგორითმი და შესაბამისად მისი გამოყენება ბერნულის გრაფებისთვის არამიზანშეწონილია. მეორეს მხვრივ, შემთხვევით გეომეტრიულ გრაფებში მის მიერ ნაპოვნი ზღვრის გამოყენებით A* ალგორითმის მნიშვნელოვანი დასწრაფება ვერ მოხერხდა.

თუმცა აღნიშნული მეთოდი მაინც აღმოჩნდა ერთი კუთხით საინტერესო. გარკვეული სიმკვრივის შემთხვევითი გეომეტრიული გრაფებისთვის იგი ხშირად ახერხებს უმოკლესი გზის ან მასთან სიგრძით საკმაოდ ახლო მარშრუტის მოძებნას და ამას A* ალგორითმზე ბევრად უფრო სწრაფად აკეთებს.

მაგალითად, 1000-წვეროიან ბრტყელ გრაფებში r=0.075 პარამეტრისთვის იგი A*-თან შედარებით 3.5-ჯერ ნაკლები წვეროს დამუშავებას ახდენს და 98%-იანი ალბათობით პოულობს უმოკლეს გზაზე არაუმეტეს 10%-ით უფრო გრძელ მარშრუტს. 5000-წვეროიან ბრტყელ გრაფებში r=0.03 პარამეტრისთვის ანალოგიურ მარშრუტებს ეს მეთოდი 95% შემთხვევებში პოულობს და ამას 8-ჯერ ნაკლები წვეროს განხილვის შედეგად ახერხებს, ვიდრე A* ძებნას სჭირდება ოპტიმალური გზის დასადგენად. 5000-წვეროიან გრაფებში d=3 და r=0.1 პარამეტრებისთვის ეს ალგორითმი ანალოგიურ მარშრუტს 91% შემთხვევაში პოულობს და ამას A* ალგორითმზე საშუალოდ 9-ჯერ უფრო სწრაფად აკეთებს.

ერთ-ერთი ყველაზე შთამბეჭდავი შედეგი ალგორითმს 50,000-წვეროიან ბრტყელ გრაფებზე აქვს. r=0.01 შემთხვევაში 99% შემთხვევებში MINIMUM-ANGLE-DFS პოულობს უმოკლეს გზაზე არაუმეტეს 10%-ით გრძელ მარშრუტს და ამას 20-ჯერ უფრო ნაკლები წვეროს განხილვით ახერხებს, ვიდრე A* ალგორითმი. r=0.02 შემთხვევაში იგი ანალოგიურ შედეგს აღწევს A*-ზე 11-ჯერ ნაკლები წვეროს განხილვით.

ამრიგად, აღნიშნული მეთოდი გარკვეულ შემთხვევებში შეიძლება გამოდგეს ხალვათ გრაფებში მოკლე მარშრუტის სწრაფად მოსაძებნად.

ლიტერატურის მიმოხილვა

[1] P. Erdõs, A. Rényi: On random graphs, I., Publ. Math. Debrecen 6 (1959), 290--297 (MR22 #10924; Zentralblatt 92,157.

[2] Random Plane Networks, E. N. Gilbert, J. SIAM, 9 (December 1961), pp. 533-543, Monograph 4121.

[3] Atkinson, M.D., J.-R. Sack, N. Santoro, and T. Strothotte (1 October 1986). "Min-max heaps and generalized priority queues.". Programming techniques and Data structures. Comm. ACM, 29(10): 996–1000.

[4] Fredman, Michael Lawrence; Tarjan, Robert E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms". Journal of the Association for Computing Machinery 34 (3): 596–615.

[5] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik* 1: 269–271

[6] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGraw-Hill. pp. 595–601. ISBN 0-262-03293-7.

[7] Robert Sedgewick, Jeffrey Scott Vitter (1986). „Shortest paths in euclidean graphs“. Algorithmica, November 1986, Volume 1, Issue 1-4, pp 31-48.

[8] I. Pohl. Bi-directional Search. In Machine Intelligence, volume 6, pages 124{140. Edinburgh Univ. Press, Edinburgh, 1971.

[9] Marsaglia, George (July 2003). "Xorshift RNGs". Journal of Statistical Software. Vol. 8 (Issue 14).

[10] Graham, Ronald; Knuth, Donald E.; Patashnik, Oren (1989), Concrete Mathematics (2nd ed.), Addison-Wesley, pp. 258–264, ISBN 978-0-201-55802-9

[11] Bentley, Jon Louis (1975), A survey of techniques for fixed-radius near neighbor searching, Technical Report SLAC-186 and STAN-CS-75-513, Stanford Linear Accelerator Center.

დანართი - სიმულაციის სტატისტიკა

ყოველი ცხრილის პირველ სტრიქონში მოცემულია გრაფში წვეროების რაოდენობა n, წიბოს არსებობის ალბათობა p, სივრცის განზომილება d, დაგენერირებულ გრაფებში წიბოების საშუალო რაოდენობა m და შერჩეულ წყვილებს შორის უმოკლესი გზის საშუალო სიგრძე l.

ცხრილების სვეტებია:

- Algorithm - ალგორითმის დასახელება, რომლის სტატისტიკური მონაცემებია შეყვანილი შესაბამის სტრიქონში
- Vertices processed on average - წვეროების საშუალო რაოდენობა, რომლებისთვისაც სრულდებოდა ალგორითმის იტერაციები ერთი ძებნის ფარგლებში
- Total vertices processed / total shortest paths length ratio – Vertices processed on average და l პარამეტრების შეფარდება
- Average ratio - საშუალო შეფარდება წვეროების იმ რაოდენობის, რომლისთვისაც ალგორითმმა იტერაციები შეასრულა და იმ რაოდენობის, რომლებიც უმოკლეს გზაში იყო შესული ერთი ძებნის ფარგლებში
- Edges examined on average - წიბოების საშუალო რაოდენობა, რომლებიც განიხილა ალგორითმმა (იმ წვეროების მეზობელთა რაოდენობების ჯამი, რომლებისთვისაც ალგორითმმა იტერაცია შეასრულა) ერთი ძებნის ფარგლებში. შევნიშნოთ, რომ ერთიდაიგივე წიბო შეიძლება ორჯერ იყოს ჩათვლილი ერთი ბოლოდან და მეორე ბოლოდან განხილვის დროს.
- Decrease-keys performed on average - ალგორითმის მიერ შესრულებული გასაღების შემცირების ოპერაციების საშუალო რაოდენობა ერთი ძებნის ფარგლებში

Bernoulli, n=500, p=0.01, d=2, m=1186.3, l=5.0396					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths	Average ratio	Edges examined on average	Decrease-keys performed

		length ratio			on average
Dijkstra	247.168	49.0451	47.9648	1294.62	485.008
A*	100.715	19.9847	18.6398	544.601	289.077
Bi-Dijkstra	33.3524	6.6180	6.5455	187.944	146.529

Bernoulli, n=500, p=0.01, d=3, m=1186.19, l=4.654					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	251.569	54.0543	51.9216	1320.68	472.655
A*	100.841	21.6675	19.8524	550.898	286.886
Bi-Dijkstra	32.314	6.9432	6.7954	183.479	143.259

Bernoulli, n=500, p=0.01, d=4, m=1186.43, l=4.445					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	251.736	56.6335	53.6303	1319.58	457.682
A*	103.756	23.3421	21.0806	566.018	285.397
Bi-Dijkstra	31.699	7.1313	6.8938	179.176	139.591

Bernoulli, n=500, p=0.02, d=2, m=2376.08, l=4.06426					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	249.168	61.3071	61.8131	2482.47	662.044
A*	52.1396	12.8288	12.2314	532.314	305.008
Bi-Dijkstra	29.7181	7.3120	7.4229	308.451	249.763

Bernoulli, n=500, p=0.02, d=3, m=2375.8, l=3.584

Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	248.005	69.1978	66.2885	2473.54	606.789
A*	52.777	14.7257	13.4408	541.018	300.097
Bi-Dijkstra	27.439	7.6559	7.4378	284.197	231.098

Bernoulli, n=500, p=0.02, d=4, m=2375.96, l=3.454					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	255.667	74.0205	71.4985	2549.51	594.269
A*	55.064	15.9420	14.6559	565.825	307.62
Bi-Dijkstra	27.631	7.9997	7.8070	286.408	232.91

Bernoulli, n=500, p=0.05, d=2, m=6236.92, l=3.23699					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	249.131	76.9637	84.4909	6279.66	919.114
A*	20.0222	6.1854	5.8572	513.168	314.365
Bi-Dijkstra	45.6637	14.1068	15.5603	1174.32	698.37

Bernoulli, n=500, p=0.05, d=3, m=6224.75, l=2.885					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	241.673	83.7688	85.1518	6114.23	817.526
A*	18.832	6.5275	6.0949	483.576	298.7
Bi-Dijkstra	25.266	8.7577	8.9160	649.797	484.266

Bernoulli, n=500, p=0.05, d=4, m=6241.5, l=2.683					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	251.069	93.5777	92.174	6373.44	769.504
A*	19.876	6.5275	6.8724	511.72	309.323
Bi-Dijkstra	23.252	8.6664	8.5795	599.454	457.615

Bernoulli, n=500, p=0.1, d=2, m=12477, l=2.72371					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	249.92	93.5777	106.419	12498.4	1107.02
A*	9.9231	3.6432	3.4011	501.402	315.953
Bi-Dijkstra	93.8517	34.4573	40.2846	4726.63	1306.43

Bernoulli, n=500, p=0.1, d=3, m=12472.5, l=2.456					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	244.381	99.5036	105.36	12248.6	952.84
A*	9.91	4.0350	3.7125	499.797	311.39
Bi-Dijkstra	33.129	13.4890	14.3797	1676.63	882.514

Bernoulli, n=500, p=0.1, d=4, m=12472.3, l=2.3					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	253.034	110.0147	111.285	12708.2	885.846
A*	10.208	4.4382	4.1565	517.074	318.414

Bi-Dijkstra	24.289	10.5604	10.7558	1231.43	763.159

Bernoulli, n=500, p=0.2, d=2, m=24951.1, l=2.22967					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	248.892	111.627	132.423	24847.8	1236.57
A*	4.97501	2.2312	2.0552	499.105	319.97
Bi-Dijkstra	140.773	63.1361	75.1479	14073.9	1816

Bernoulli, n=500, p=0.2, d=3, m=24942.3, l=2.118					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	251.11	118.56	132.223	25076.5	1062.63
A*	5.118	2.4164	2.2324	513.152	323.767
Bi-Dijkstra	57.682	27.2342	29.9551	5782.44	1384.59

Bernoulli, n=500, p=0.2, d=4, m=24978.8, l=2.015					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	250.791	124.462	130.302	25110.2	979.347
A*	4.949	2.45608	2.27067	497.296	320.044
Bi-Dijkstra	31.374	15.5702	16.2108	3153.2	1176.19

Bernoulli, n=5000, p=0.001, d=2, m=11874.3, l=6.99806					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths	Average ratio	Edges examined on average	Decrease-keys performed

		length ratio			on average
Dijkstra	2476.83	353.931	347.769	13002.2	4875.36
A*	1007.22	143.929	136.852	5461.2	2894.63
Bi-Dijkstra	109.512	15.6489	15.5016	626.043	507.039

Bernoulli, n=5000, p=0.001, d=3, m=11874.3, l=6.443					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2544.21	394.88	382.114	13355.1	4772.02
A*	1050.09	162.982	152.683	5715.95	2915.22
Bi-Dijkstra	107.841	16.7377	16.4097	615.836	498.268

Bernoulli, n=5000, p=0.001, d=4, m=11874.3, l=6.162					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2444.42	396.693	378.145	12837.7	4535.57
A*	1001.25	162.487	148.568	5467.59	2754.9
Bi-Dijkstra	103.319	16.7671	16.2279	591.502	478.601

Bernoulli, n=5000, p=0.002, d=2, m=23751, l=5.68378					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2498.62	439.606	441.825	24932.3	6718.01
A*	525.016	92.3709	89.9396	5369.05	3071.41
Bi-Dijkstra	95.2248	16.7538	16.8688	996.525	869.634

Bernoulli, n=5000, p=0.002, d=3, m=23751, l=5.035

Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2530.39	502.56	497.539	25252.2	6268.27
A*	531.715	105.604	101.232	5463.16	3050.31
Bi-Dijkstra	90.791	18.032	17.9403	948.668	828.163

Bernoulli, n=5000, p=0.002, d=4, m=23750.9, l=4.673					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2479.91	530.688	515.936	24786	5947
A*	518.598	110.978	104.417	5345.58	2949.06
Bi-Dijkstra	86.38	18.4849	18.1714	905.156	790.732

Bernoulli, n=5000, p=0.005, d=2, m=59394.2, l=4.80024					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2503.38	521.512	546.055	60394.4	9468.62
A*	210.801	43.9147	43.0562	5166.73	3168.68
Bi-Dijkstra	115.244	24.0079	25.0709	2846.72	2409.93

Bernoulli, n=5000, p=0.005, d=3, m=59395, l=4.076					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2488.18	610.447	610.272	60259.8	8365.45
A*	207.373	50.8766	48.8482	5101.73	3076.41
Bi-Dijkstra	78.8	19.3327	19.3607	1949.11	1732.28

Bernoulli, n=5000, p=0.005, d=4, m=59394.2, l=3.699					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2527.09	683.182	667.738	61210.5	7773.1
A*	208.583	56.389	52.9288	5136.59	3083.72
Bi-Dijkstra	74.627	20.1749	19.8575	1843.24	1642.58

Bernoulli, n=5000, p=0.01, d=2, m=118852, l=4.28122					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2500.36	584.031	641.33	119364	12035.7
A*	105.144	24.5593	23.9819	5077.89	3209.9
Bi-Dijkstra	294.132	68.7029	75.8903	14220.6	7218.31

Bernoulli, n=5000, p=0.01, d=3, m=118850, l=3.634					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2541.59	699.392	715.121	121745	10163.3
A*	109.488	30.1288	29.1452	5297.38	3275.29
Bi-Dijkstra	90.209	24.8236	25.4311	4374.3	3584.05

Bernoulli, n=5000, p=0.01, d=4, m=118854, l=3.231					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2471.89	765.054	754.973	118586	9050.7
A*	104.976	32.4903	30.8803	5080.5	3128.15

Bi-Dijkstra	70.188	21.7233	21.5801	3401.8	2904.29

Bernoulli, n=5000, p=0.05, d=2, m=624860, l=3.15049					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2500.82	793.786	928.927	625108	19378.6
A*	20.0273	6.3569	6.04682	5019.76	3270.04
Bi-Dijkstra	1462.22	464.126	544.282	365627	27519.1

Bernoulli, n=5000, p=0.05, d=3, m=624884, l=2.793					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2517.93	901.515	1029.25	629599	14714.8
A*	20.194	7.23022	6.7617	5060.35	3235.42
Bi-Dijkstra	443.223	158.691	181.562	111021	17052.7

Bernoulli, n=5000, p=0.05, d=4, m=624925, l=2.555					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2509.67	982.259	1046.9	627815	12663.8
A*	20.381	7.97691	7.59645	5113.5	3238.22
Bi-Dijkstra	143.335	56.0998	58.7715	35957.3	12537.1

Bernoulli, n=5000, p=0.1, d=2, m=1249720, l=2.69225					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths	Average ratio	Edges examined on average	Decrease-keys performed

		length ratio			on average
Dijkstra	2501.48	929.141	1101.35	1250480	21296.5
A*	10.0105	3.71828	3.48663	5010.19	3259.06
Bi-Dijkstra	1616.02	600.251	712.116	807887	33604.3

Bernoulli, n=5000, p=0.1, d=3, m=1249790, l=2.415					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2532.54	1048.67	1184.95	1266150	16286.5
A*	10.073	4.17101	3.87282	5042.25	3240.41
Bi-Dijkstra	771.867	319.614	362.446	386072	23042

Bernoulli, n=5000, p=0.1, d=4, m=1249700, l=2.272					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2451.21	1078.88	1168.4	1225590	13807
A*	9.839	4.33055	4.09967	4926.1	3224.11
Bi-Dijkstra	260.745	114.765	123.663	130497	17710.2

Bernoulli, n=50000, p=0.0005, d=2, m=593769, l=6.2405					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	24912.4	3992.05	4110.55	602085	95514
A*	2106.73	337.591	335.733	51649.7	31673.5
Bi-Dijkstra	316.875	50.7772	52.0708	7839.17	7283.94

Bernoulli, n=50000, p=0.0005, d=3, m=593772, l=5.23

Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25047.2	4789.15	4746.01	606262	84877.1
A*	2027.9	387.744	370.558	49826.1	30178.8
Bi-Dijkstra	242.79	46.4226	46.88	6001.58	5622.43

Bernoulli, n=50000, p=0.0005, d=4, m=593768, l=4.6					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	24485.6	5322.95	5242.12	593505	76971.7
A*	2111.25	458.967	440.65	51975.2	30589.9
Bi-Dijkstra	233.45	50.75	50.0698	5778.25	5408.45

Bernoulli, n=50000, p=0.001, d=2, m=1187600, l=5.6735					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25130.9	4429.53	4706.05	1200320	122675
A*	1063.9	187.521	186.068	51363.9	32433.7
Bi-Dijkstra	702.373	123.799	132.187	34037.9	26590.2

Bernoulli, n=50000, p=0.001, d=3, m=1187600, l=4.57					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	24707.4	5406.44	5425.7	1184270	102386
A*	1099.28	240.543	227.778	53167.4	32115.9
Bi-Dijkstra	261.79	57.2845	57.7026	12697.5	11817.7

Bernoulli, n=50000, p=0.001, d=4, m=1187600, l=4.18					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25508.5	6102.5	6018.65	1223030	92811.7
A*	1084.63	259.481	247.158	52506.2	32697
Bi-Dijkstra	227.52	54.4306	53.8759	11035.4	10333.5

Bernoulli, n=50000, p=0.002, d=2, m=2375460, l=5.1888					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25112.8	4839.82	5281.46	2389260	157654
A*	525.141	101.207	100.098	50306.7	32615.9
Bi-Dijkstra	3518.15	678.028	741.981	336559	98574.3

Bernoulli, n=50000, p=0.002, d=3, m=2375460, l=4.52					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25754.8	5697.97	5868.85	2455280	125356
A*	539.17	119.285	117.847	51700.6	33676.2
Bi-Dijkstra	384.08	84.9735	85.5452	36883.5	30403

Bernoulli, n=50000, p=0.002, d=4, m=2375450, l=3.77					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25358.7	6726.43	6926.02	2419790	109010
A*	467.01	123.875	123.406	44833.2	30277.5

Bi-Dijkstra	222.7	59.0716	61.1004	21416.9	19483.9

Bernoulli, n=50000, p=0.005, d=2, m=5940510, l=4.5918					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	24951	5433.82	6087.84	5929650	217219
A*	206.476	44.9663	44.164	49226.1	32812.9
Bi-Dijkstra	11338.6	2469.32	2779.08	2696470	247218

Bernoulli, n=50000, p=0.005, d=3, m=5940510, l=3.72					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	24982.6	6715.75	7510.31	5939840	155828
A*	215.36	57.8925	56.8225	51353.1	33439.3
Bi-Dijkstra	1426.08	383.355	418.096	340018	105356

Bernoulli, n=50000, p=0.005, d=4, m=5940500, l=3.36					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25027	7448.52	7668.96	5954190	132511
A*	228.27	67.9375	65.5347	54444.9	34089.3
Bi-Dijkstra	317.22	94.4107	96.6435	75709.3	55054.1

Bernoulli, n=50000, p=0.01, d=2, m=11887300, l=4.142					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths	Average ratio	Edges examined on average	Decrease-keys performed

		length ratio			on average
Dijkstra	24805.5	5988.77	6747.02	11795000	265908
A*	103.508	24.9898	24.4477	49294	32963.5
Bi-Dijkstra	14638.5	3534.17	3977.43	6961520	365751

Bernoulli, n=50000, p=0.01, d=3, m=11887300, l=3.52					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	23113.4	6566.3	7144.08	10992100	180465
A*	108.79	30.9063	29.6105	51807.2	34271.7
Bi-Dijkstra	2844.62	808.131	835.364	1354350	175131

Bernoulli, n=50000, p=0.01, d=4, m=11887300, l=3.06					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	24366.4	7962.89	8207.03	11590700	149423
A*	87.98	28.7516	28.2652	41912.4	29004.7
Bi-Dijkstra	612.24	200.078	203.704	291667	112640

Geometric, n=500, r=0.05, d=2, m=939.667, l=11.1285					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	64.9994	5.84081	5.34077	286.446	80.3915
A*	32.6511	2.93401	2.24277	147.838	49.2875
Bi-Dijkstra	53.2902	4.78863	4.25297	238.333	71.8298

Geometric, n=500, r=0.1, d=2, m=3589.1, l=7.43841

Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	249.478	33.5392	31.1064	3685.34	478.319
A*	22.8454	3.07127	2.63071	350.734	104.998
Bi-Dijkstra	165.697	22.2758	19.9443	2460.45	378.067

Geometric, n=500, r=0.2, d=2, m=13132.1, l=3.70988					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	249.276	67.1924	62.7642	13727.2	769.812
A*	6.65011	1.79254	1.59974	390.521	149.981
Bi-Dijkstra	167.09	45.0391	40.5095	9212.12	715.503

Geometric, n=500, r=0.2, d=3, m=3298.32, l=4.949					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	244.698	49.4439	45.2467	3428.2	459.79
A*	16.971	3.42918	2.93601	263.033	92.243
Bi-Dijkstra	104.235	21.0618	18.7758	1489.02	296.661

Geometric, n=500, r=0.2, d=4, m=742.34, l=11.775					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	186.05	15.8004	14.8095	686.65	224.157
A*	92.052	7.81758	6.45073	353.988	129.408
Bi-Dijkstra	85.015	7.21996	6.60381	330.7	122.214

Geometric, n=500, r=0.4, d=4, m=43028, l=1.97491					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	249.323	126.245	120.397	45729.3	999.424
A*	2.40788	1.21924	1.1533	466.702	264.703
Bi-Dijkstra	167.377	84.7516	77.9559	30561.5	1197.35

Geometric, n=500, r=0.4, d=3, m=20241.3, l=2.396					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	248.287	103.626	96.5303	21659.4	839.393
A*	3.45	1.4399	1.33518	339.554	173.866
Bi-Dijkstra	109.863	45.8527	41.1638	9648.61	832.382

Geometric, n=500, r=0.4, d=3, m=8682.78, l=2.836					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	245.214	86.4647	78.9998	9276.92	641.415
A*	6.223	2.19429	1.93937	276.872	123.807
Bi-Dijkstra	75.188	26.512	23.6136	2925.32	494.224

Geometric, n=5000, r=0.015, d=2, m=8714.92, l=13.63					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	80.0498	5.87305	5.20263	333.893	96.1929
A*	43.1981	3.16934	2.33111	184.681	61.0799

Bi-Dijkstra	66.9759	4.91386	4.21391	283.047	86.1427

Geometric, n=5000, r=0.02, d=2, m=15432.6, l=43.612					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2456.65	56.3297	51.9496	15442.4	3206.67
A*	584.598	13.4045	11.1755	3754.53	967.833
Bi-Dijkstra	1515.97	34.7605	30.9171	9597.42	2056.89

Geometric, n=5000, r=0.03, d=2, m=34439.2, l=24.2493					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2497.46	102.991	95.1268	34721.2	4329.02
A*	209.562	8.64197	7.19363	2981.31	674.548
Bi-Dijkstra	1651.92	68.1224	60.3197	22999.4	3040.71

Geometric, n=5000, r=0.05, d=2, m=94038.7, l=14.2653					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2494.38	174.857	161.807	95149.8	6138.55
A*	76.5586	5.36677	4.51166	2992.13	664.726
Bi-Dijkstra	1665.84	116.776	103.657	63598.7	4543.28

Geometric, n=5000, r=0.05, d=3, m=6183.72, l=17.947					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths	Average ratio	Edges examined on average	Decrease-keys performed

		length ratio			on average
Dijkstra	147.052	8.19368	6.77072	485.818	164.019
A*	93.463	5.20772	3.7092	314.279	113.994
Bi-Dijkstra	104.128	5.80197	4.70682	350.709	124.281

Geometric, n=5000, r=0.1, d=2, m=359844, l=7.22293					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2497.29	345.744	320.753	368917	9080.9
A*	21.5286	2.98059	2.56403	3326.7	884.75
Bi-Dijkstra	1672.01	231.486	206.088	247200	7418.83

Geometric, n=5000, r=0.1, d=3, m=46656.5, l=9.453					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2514.43	265.993	243.431	48356	4623.8
A*	89.481	9.46588	7.95734	1846.11	472.947
Bi-Dijkstra	1041.41	110.167	96.3362	20197.8	2453.45

Geometric, n=5000, r=0.1, d=4, m=5360.42, l=33.788					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	683.6	20.232	18.3495	2039.51	737.805
A*	486.838	14.4086	11.9703	1467.49	549.817
Bi-Dijkstra	364.542	10.7891	9.40825	1115.27	416.614

Geometric, n=5000, r=0.2, d=2, m=1313560, l=3.72237

Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2502.95	672.408	628.75	1375840	12427.1
A*	6.66779	1.79128	1.60438	3924.55	1470.73
Bi-Dijkstra	1672.2	449.23	403.666	918735	11804.3

Geometric, n=5000, r=0.3, d=3, m=330653, l=4.547					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2466.14	542.367	542.367	342770	8093.01
A*	14.648	3.22146	2.80268	2297.74	700.362
Bi-Dijkstra	1074.12	236.227	206.289	150542	5868.3

Geometric, n=5000, r=0.3, d=4, m=74243.2, l=5.601					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	2482.26	443.181	405.568	78106.9	5281.57
A*	44.675	7.97625	6.7815	1621.68	476.44
Bi-Dijkstra	680.397	121.478	106.351	21871.8	2549.69

Geometric, n=50000, r=0.005, d=2, m=97823.4, l=35.278					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	369.953	10.4868	9.04758	1623.41	436.564
A*	215.115	6.0977	4.40478	955.784	281.822
Bi-Dijkstra	298.726	8.46776	6.96772	1321.88	362.71

Geometric, n=50000, r=0.01, d=2, m=389490, l=71.597					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	24966.6	348.71	322.127	390088	43849.5
A*	1753.88	24.4965	20.1068	27650.6	5333.62
Bi-Dijkstra	16622	232.161	205.225	259783	29816.1

Geometric, n=50000, r=0.015, d=2, m=872169, l=47.2109					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25134.3	532.384	492.584	880552	57528.1
A*	776.895	16.4558	13.5958	27474.6	4684.55
Bi-Dijkstra	16783.8	355.507	315.395	588191	39726.9

Geometric, n=50000, r=0.025, d=2, m=2402380, l=28.2885					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	25070.8	886.256	818.692	2426510	75623.5
A*	285.446	10.0905	8.36614	27986.7	4529.48
Bi-Dijkstra	16744.1	591.904	524.27	1621230	53574.3

Geometric, n=50000, r=0.05, d=2, m=9402860, l=14.1451					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	24815.2	1754.33	1621.15	9463350	102413
A*	75.9192	5.36717	4.50787	29728.8	5513.52

Bi-Dijkstra	16573.5	1171.68	1038.37	6322600	76895.7

Geometric, n=50000, r=0.05, d=3, m=618743, l=18.76					
Algorithm	Vertices processed on average	Total vertices processed / total shortest paths length ratio	Average ratio	Edges examined on average	Decrease-keys performed on average
Dijkstra	26080.7	1390.23	1272.32	655776	47910.7
A*	654.23	34.8737	28.193	17056.5	3187.44
Bi-Dijkstra	10988.8	585.756	509.006	276523	22958.1

Minimum-angle-DFS, Geometric, n=1000, d=2									
p	Average number of edges	Exact	Within 10%	Within 20%	Within 30%	Within 40%	>1.4 * shortest	MADFS visited vertices	A* visited vertices
0.05	3760.25	7.345%	44.205%	15.956%	8.151%	5.05%	19.293%	24.4523	95.133
0.075	8276.77	15.83%	82.408%	1.37%	0.194%	0.075%	0.123%	10.6844	37.5778
0.1	14383	23.984%	75.959%	0.052%	0.003%	0.001%	0.001%	7.86996	21.8383
0.15	30926.4	39.864%	60.136%	0%	0%	0%	0%	5.27806	10.6875

Minimum-angle-DFS, Geometric, n=1000, d=3									
0.15	5933.01	23.606%	53.876%	14.512%	4.31%	1.54%	2.156%	7.53709	36.1281
0.2	13235.4	38.138%	59.575%	2.092%	0.147%	0.036%	0.012%	5.05068	15.8047
0.25	24252.4	49.392%	50.434%	0.171%	0.002%	0.001%	0%	3.97594	9.136

Minimum-angle-DFS, Geometric, n=5000, d=2									
0.02	15440.3	1.257%	11.821%	12.22%	11.141%	9.35%	54.211%	95.4626	581.788
0.03	34450.5	3.208%	91.742%	3.929%	0.602%	0.254%	0.265%	26.9796	210.593
0.05	94050.7	8.085%	91.915%	0%	0%	0%	0%	15.4374	76.9197
0.1	359817	23.986%	76.014%	0%	0%	0%	0%	7.81849	21.6589

Minimum-angle-DFS, Geometric, n=5000, d=3									
0.1	46667.4	12.338%	78.56%	8.111%	0.754%	0.138%	0.099%	10.2226	90.836
0.15	148447	25.977%	73.975%	0.045%	0.003%	0%	0%	6.43681	29.783

0.2	331003	38.641%	61.359%	0%	0%	0%	0%	4.82124	14.5374

Minimum-angle-DFS, Geometric, n=50000, d=2									
0.01	389365	0.415%	98.725%	0.753%	0.073%	0.018%	0.016%	78.6623	1758.88
0.02	1544160	1.58%	98.42%	0%	0%	0%	0%	37.8993	441.47
0.03	3444800	3.35%	96.65%	0%	0%	0%	0%	25.3017	201.029
0.05	9404710	8.236%	91.764%	0%	0%	0%	0%	15.2951	76.1233