Алгоритм шинжилгээ, зохиомж 2

**Genetic algorithm, Critical nodes тайлан**

*Оюутан: Т.Билгүүн, Д.Балжинням, Б.Бадрангийх, Г.Батзориг*

Genetic algorithm нь байгалийн шалгарал, генетикийн үйл явцаас сэдэвлэсэн хайлт, оновчлолын арга юм. Уг алгоритм нь өгөгдсөн асуудлын оновчтой шийдлийг олоход хэрэглэгддэг.

Genetic algorithm-д боломжит шийдлүүдийн population (популяци буюу individuals эсвэл хромосом гэж нэрлэдэг) хамгийн сайн шийдлийг олохын тулд үе дамждаг. Алгоритм нь нөхөн үржихүй (reproduction), кроссовер, мутаци зэрэг биологийн процессуудыг дуурайж, популяцийг давталттайгаар сайжруулдаг.

**Genetic algorithm хэрхэн ажилладаг вэ?**

**Initialization (эхлэл):** Алгоритм нь санамсаргүй хүмүүсийн анхны популяци үүсгэх замаар эхэлдэг. Хувь хүн бүр асуудлын боломжит шийдлийг төлөөлдөг.

**Evaluation (үнэлгээ):** Популяцийн хувь хүн бүрийг шийдлийн чанарыг тодорхойлдог фитнессийн функцийг ашиглан үнэлдэг. Фитнессийн функц нь асуудлын зорилго эсвэл зорилгыг тодорхойлдог.

**Selection (сонголт):** Хувь хүмүүсийг бие бялдрын хувьд одоогийн хүн амын дундаас сонгоно. Өндөр бие бялдартай хүмүүс нөхөн үржихүйд сонгогдох магадлал өндөр байдаг. Энэ үйл явц нь ихэвчлэн тэмцээн сонгох эсвэл рулет дугуй сонгох зэрэг арга техникийг ашиглан хэрэгждэг.

**Reproduction (нөхөн үржихүй):** Сонгогдсон бодгальуудыг дараагийн хойч үедээ үр удмаа бий болгоход ашигладаг. Энэ нь кроссовер болон мутацийн үйлдлээр хийгддэг.

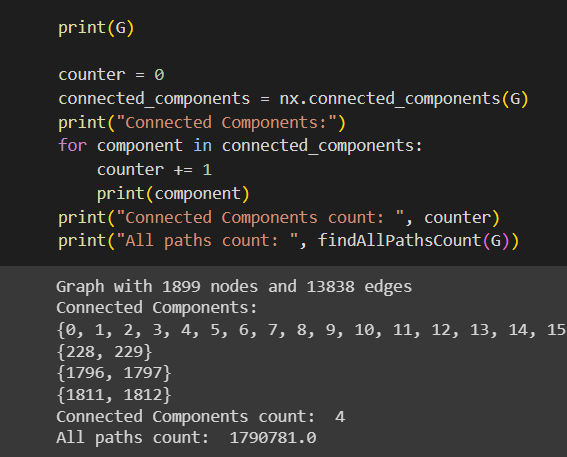
* Кроссовер: Эцэг эхийн хоёр хувь нэгдэж, генетикийн мэдээлэл солилцох замаар нэг буюу хэд хэдэн үр удмыг бий болгодог. Энэ нь бэлгийн нөхөн үржихүйн үйл явцыг дуурайдаг бөгөөд эцэг эхийн хооронд шинж чанар солилцдог. Кроссовер цэг нь генетикийн материалыг хаана солилцохыг тодорхойлдог.
* Мутаци: Ургийн удамшлын материалд санамсаргүй байдлаар бага зэрэг өөрчлөлт ордог. Энэ нь генетикийн олон янз байдлыг нэвтрүүлж, алгоритмыг орон нутгийн оновчтой байдалд оруулахаас сэргийлдэг.

**Replacement (орлуулах):** Үр удмын популяци нь алгоритмын загвараас хамааран одоогийн популяцийг бүхэлд нь эсвэл хэсэгчлэн орлоно. Дараагийн үе нь бүрэлдэж байна.

**Termination (дуусгах):** Алгоритм нь тодорхой тооны үеийн туршид эсвэл дуусгавар болох шалгуурыг хангах хүртэл үнэлгээ, сонгох, хуулбарлах, солих алхмуудыг давтдаг. Дуусгах шалгуур нь бие бялдрын чийрэгжилтийн хангалттай түвшинд хүрэх, эсвэл урьдчилан тодорхойлсон хугацааны хязгаар байж болно.

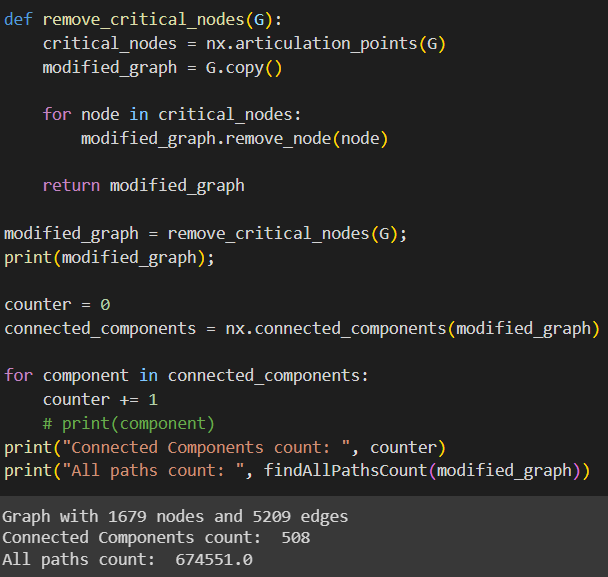
***OCLinks.txt файлд байгаа графын мэдээлэл***

Оройн тоо 1899, ирмэгийн тоо 13939 ширхэг байсан. Үүнээс connected component нь нийт 4 ширхэг байсан.



***Networkx сангийн функц ашиглах нь***

Энэхүү зурагт үзүүлснээр бид networkx сангийн articulation\_points() функцыг ашиглан critical nodes-үүдээ олох боломжтой. Мөн connected\_components() функцээр хуваагдсан sub graph-уудаа тоолох боломжтой юм.



***Articulation point***

График дахь articulation цэгүүдийг олох алгоритм нь DFS алгоритм дээр суурилж бичигдсэн функц юм. Графын оройг түүний ирмэгүүдийн хамт устгахад графикт байгаа холбогдсон компонентүүдын тоог нэмэгдүүлдэг.

***Articulation point reference***

Hopcroft, J.; Tarjan, R. (1973). “Efficient algorithms for graph manipulation”. Communications of the ACM 16: 372–378. doi:10.1145/362248.362272

***Genetic algorithm implementation (Үр дүн SEQ = 1352195.0, 6 минут)***

import random

from collections import defaultdict

def read\_graph\_from\_file(file\_path):

graph = defaultdict(set)

with open(file\_path, 'r') as f:

num\_nodes = int(f.readline().strip())

for line in f:

node, neighbors = line.strip().split(':')

node = int(node)

neighbors = set(map(int, neighbors.split()))

graph[node] = neighbors

return graph

def fitness\_function(individual, graph, memo):

individual\_key = tuple(sorted(individual))

if individual\_key in memo:

return memo[individual\_key]

num\_components = get\_number\_of\_connected\_components(graph)

graph\_copy = remove\_nodes(graph, individual)

num\_components\_after\_removal = get\_number\_of\_connected\_components(graph\_copy)

fitness = num\_components\_after\_removal - num\_components

memo[individual\_key] = fitness

return fitness

def get\_number\_of\_connected\_components(graph):

visited = set()

num\_components = 0

for node in graph:

if node not in visited:

dfs(graph, node, visited)

num\_components += 1

return num\_components

def dfs(graph, node, visited):

visited.add(node)

for neighbor in graph[node]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

def remove\_nodes(graph, nodes):

modified\_graph = graph.copy()

for node in nodes:

if node in modified\_graph:

del modified\_graph[node]

for neighbor in modified\_graph:

modified\_graph[neighbor].discard(node)

return modified\_graph

def generate\_initial\_population(graph, population\_size, num\_critical\_nodes):

nodes = list(graph.keys())

return [random.sample(nodes, num\_critical\_nodes) for \_ in range(population\_size)]

def crossover(parent1, parent2):

if len(parent1) == 1 or len(parent2) == 1:

return parent1, parent2

crossover\_point = random.randint(1, min(len(parent1), len(parent2)) - 1)

child1 = parent1[:crossover\_point] + parent2[crossover\_point:]

child2 = parent2[:crossover\_point] + parent1[crossover\_point:]

return child1, child2

def mutate(individual, mutation\_rate, graph):

nodes = list(graph.keys())

individual = [random.choice(nodes) if random.random() < mutation\_rate else i for i in individual]

return individual

def genetic\_algorithm(graph, num\_critical\_nodes, population\_size=100, generations=100, mutation\_rate=0.15):

population = generate\_initial\_population(graph, population\_size, num\_critical\_nodes)

memo = {}

best\_individual = max(population, key=lambda x: fitness\_function(x, graph, memo))

best\_fitness = fitness\_function(best\_individual, graph, memo)

for generation in range(generations):

print(generation, "th change")

new\_population = []

for \_ in range(population\_size // 2):

parent1, parent2 = random.sample(population, 2)

child1, child2 = crossover(parent1, parent2)

child1 = mutate(child1, mutation\_rate, graph)

child2 = mutate(child2, mutation\_rate, graph)

new\_population.extend([child1, child2])

population = new\_population

current\_best\_individual = max(population, key=lambda x: fitness\_function(x, graph, memo))

current\_best\_fitness = fitness\_function(current\_best\_individual, graph, memo)

if current\_best\_fitness > best\_fitness:

best\_individual = current\_best\_individual

best\_fitness = current\_best\_fitness

return best\_individual, best\_fitness

***Genetic algorithm implementation (Үр дүн 845678, 5 минут)***

***Уг алгоритмын ажиллагаа буюу pseudocode:***

1. Networkx сангийн articulation\_nodes() ашиглаад хамгийн ашигтай critical nodes-үүдээ олно.
2. Олсон оройнуудаасаа mutation функцаа авч ашиглаж байгаа. Ингэснээр random орой авч ашигласнаар хамаагүй efficient буюу үр ашигтай юм.
3. Population generate хийх
4. Evolution үйл явц
   1. Crossover хэрэгжүүлэлт
      1. Дурын хоёр дараалал сонгож аваад тэрнийхээ эхний дурын хэсгийг сонгож нөгөөгийнхөө эхний хэсэг дээр сольж тавьсан.
   2. Mutation хэрэгжүүлэлт
      1. Mutation дурын нэг дараалал сонгож тэрний дурын нэг оройг нь дурын нэг оройгоор сольж байгаа.
   3. Холболт дээр үндэслэн population-ыг ангилах

i=0

while i<iteration:

i+=1

j=0

while j<n//2:

j+=1

rand\_gene = random.randint(0, n-1)

new\_seq = population[rand\_gene].seq.copy()

l=0

while l<10:

l+=1

mut\_point = random.randint(0, k-1)

cod = random.choice(node\_list)

while cod in new\_seq:

cod = random.choice(node\_list)

new\_seq[mut\_point] = cod

new\_seq.sort()

check=0

for s in population:

if new\_seq == s.seq:

check+=1

if new\_seq in used :

check+=1

else :

used.append(new\_seq)

if check > 0 :

j -=1

else:

new\_gene = Gene(new\_seq, zor(graph.subgraph([element for element in node\_list if element not in new\_seq])))

population.append(new\_gene)

population.sort(key=lambda x: x.cc)

population = population[:n]

j=0

while j< n//10:

j+=1

rand\_gene1 = random.randint(0, n//10- 1)

rand\_gene2 = random.randint(n//10, n - 1)

crossover\_point = random.randint(0, k-1)

new\_seq = population[rand\_gene1].seq.copy()

new\_seq[crossover\_point:] = population[rand\_gene2].seq[crossover\_point:].copy()

new\_seq.sort()

check=0

for s in population:

if new\_seq == s.seq:

check+=1

if new\_seq in used :

check+=1

else :

used.append(new\_seq)

dup = {x for x in new\_seq if new\_seq.count(x) > 1}

if len(dup) > 0 :

check+=1

if check>0 :

j -=1

else :

new\_gene = Gene(new\_seq, zor(graph.subgraph([element for element in node\_list if element not in new\_seq])))

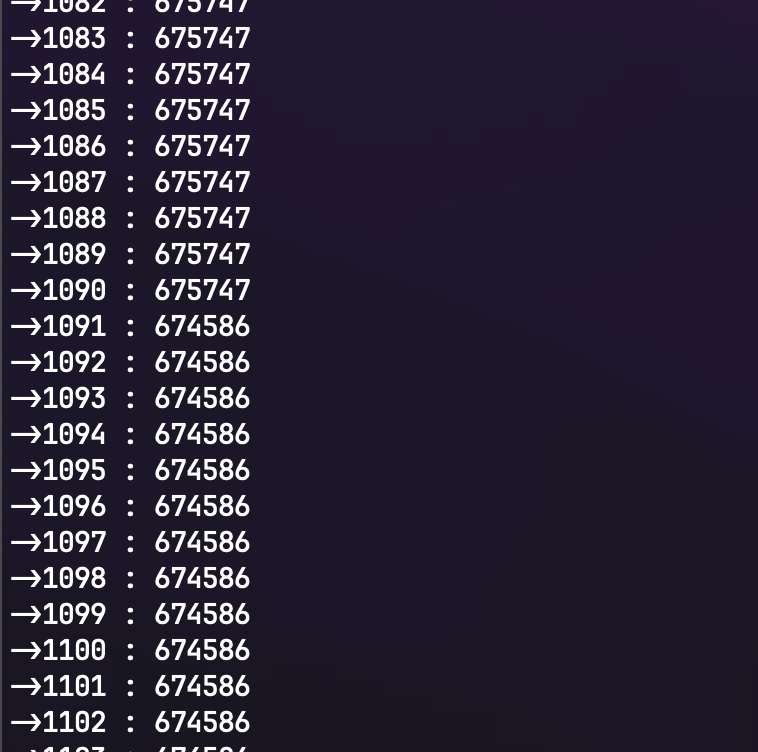
population[rand\_gene2] = new\_gene

population.sort(key=lambda x: x.cc)

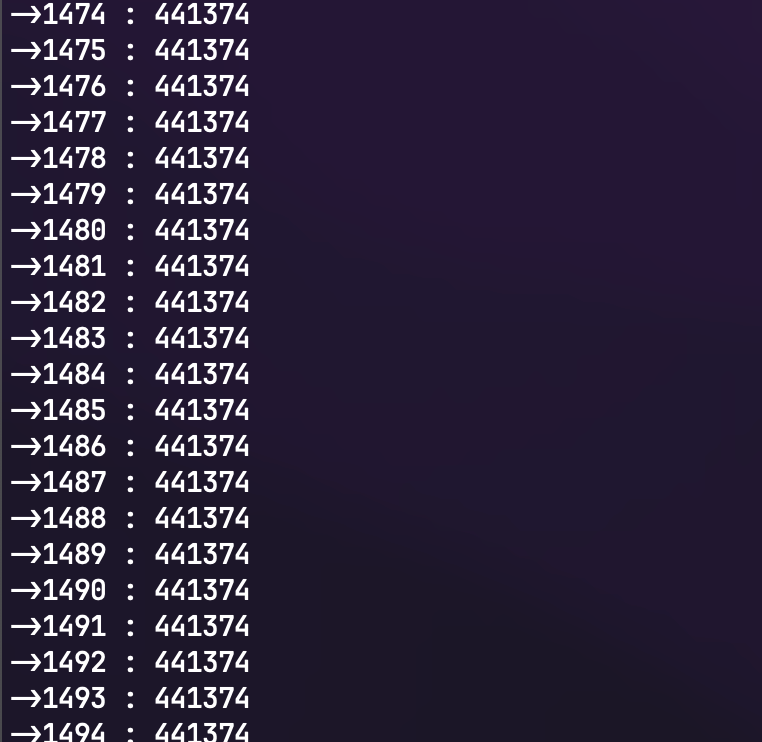
print("->%d : %d" %(i, population[0].cc))

ccs.append(population[0].cc)

***Үр дүн part 1 k =190:***

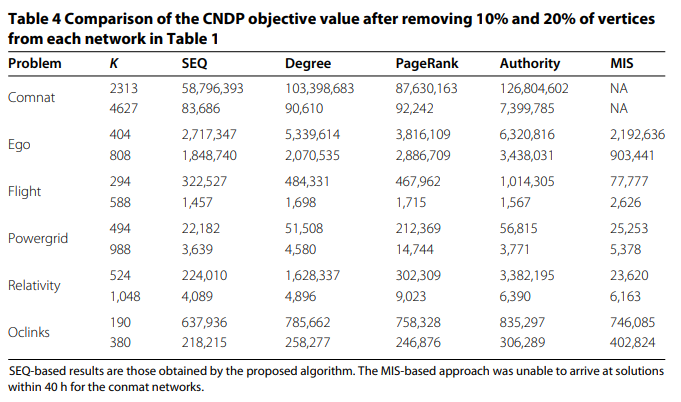


***Үр дүн part 2 k = 380:***



“Efficiently identifying critical nodes in large complex networks Mario Ventresca1\* and Dionne Aleman2” материалын үр дүнтэй харьцуулахад

Бидний гаргасан үр дүн уг үр дүнг гүйцэж чадахгүй харагдаж байна.



***Ашигласан материал***

* Evolutionary algorithms and their applications to engineering problems Adam Slowik1 • Halina Kwasnicka2
* Efficiently identifying critical nodes in large complex networks Mario Ventresca1\* and Dionne Aleman2
* [connected\_components — NetworkX 3.1 documentation](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.components.connected_components.html)
* [articulation\_points — NetworkX 3.1 documentation](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.components.articulation_points.html)