# Genetic algorithm critical node

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

# Ярих сэдэв

## 01. Networkx

Networkx-ийг ашиглах нь

## 02. Genetic algorithm

Хэрхэн генетик алгоритмыг ашигласан талаар

## 03. Үр дүн

Бичсэн кодны үр дүнг харьцуулах

## 04. Дүгнэлт

Гарч ирсэн үр дүн дээрээ дүгнэлт хийх

# 01

# Networkx

Networkx - articulation points()

# Анхны мэдээлэл

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

```python
print(G)

counter = 0
connected_components = nx.connected_components(G)
print("Connected Components:")
for component in connected_components:
    counter += 1
    print(component)
print("Connected Components count: ", counter)
print("All paths count: ", findAllPathsCount(G))
```

```
Graph with 1899 nodes and 13838 edges
Connected Components:
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
{228, 229}
{1796, 1797}
{1811, 1812}
Connected Components count:  4
All paths count:  1790781.0
```

# Networkx articulation points

```python
def remove_critical_nodes(G):
    critical_nodes = nx.articulation_points(G)
    modified_graph = G.copy()

    for node in critical_nodes:
        modified_graph.remove_node(node)

    return modified_graph

modified_graph = remove_critical_nodes(G);
print(modified_graph);

counter = 0
connected_components = nx.connected_components(modified_graph)

for component in connected_components:
    counter += 1
    # print(component)
print("Connected Components count: ", counter)
print("All paths count: ", findAllPathsCount(modified_graph))
```

```
Graph with 1679 nodes and 5209 edges
Connected Components count:  508
All paths count:  674551.0
```

## 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

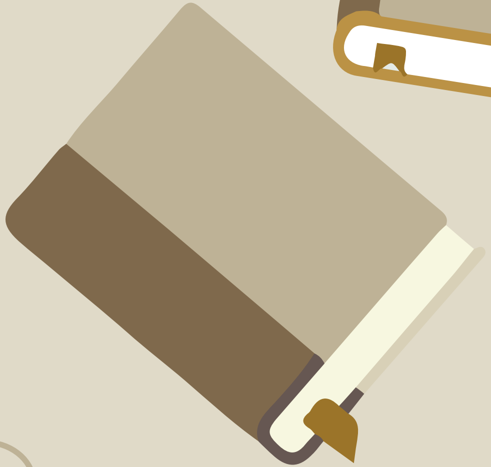


График дахь articulation цэгүүдийг олох алгоритм нь DFS алгоритм дээр суурилж бичигдсэн функц юм. Графын оройг түүний ирмэгүүдийн хамт устгахад графикт байгаа холбогдсон компонентүүдын тоог нэмэгдүүлдэг.

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

Articulation цэгүүд нь график дахь зангилааны цэг бөгөөд тэдгээрийг арилгах үед график дахь салангид хэсгүүдийн тоог нэмэгдүүлдэг.

# Ашигласан алгоритм

Genetic algorithm-ийг ашиглах нь

## *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
```

```python
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
```

```python
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

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.

# Алгоритмын ажиллагаа

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
   a. Crossover хэрэгжүүлэлт
      i. Дурын хоёр дараалал сонгож аваад тэрнийхээ эхний дурын хэсгийг сонгож нөгөөгийнхөө эхний хэсэг дээр сольж тавьсан.
   b. Mutation хэрэгжүүлэлт
      i. Mutation дурын нэг дараалал сонгож тэрний дурын нэг оройг нь дурын нэг оройгоор сольж байгаа.
   c. Холболт дээр үндэслэн population-ыг ангилах

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

```
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)
```

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

```python
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

Кодны үр дүн боловсруулалт

# Үр дүн

Үр дүн k=190 7 минут:

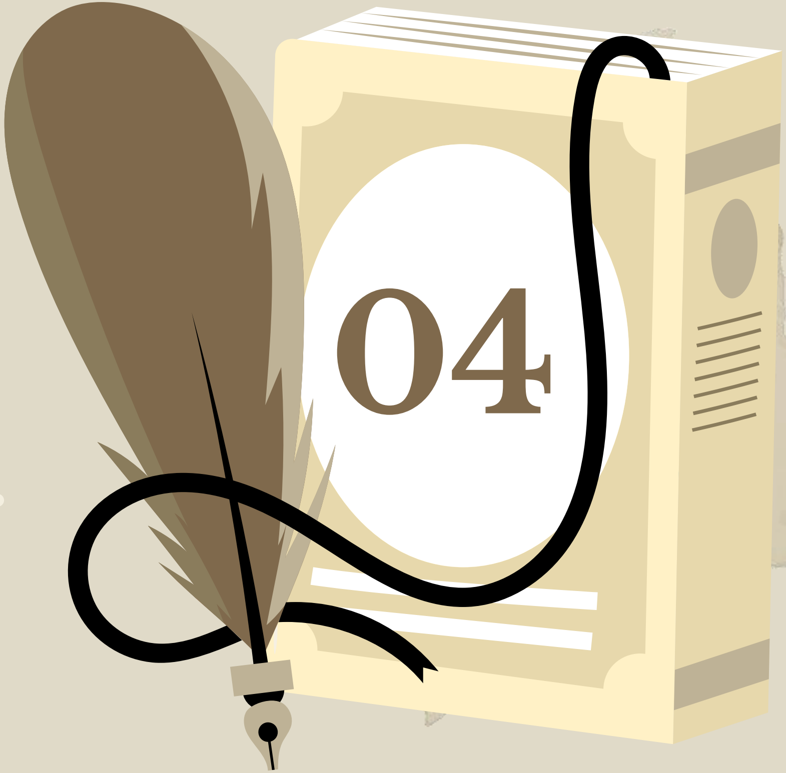
```
→1485 : 669947
→1486 : 669947
→1487 : 669947
→1488 : 668790
→1489 : 668790
→1490 : 669947
→1491 : 669947
→1492 : 669947
→1493 : 669947
→1494 : 669947
→1495 : 669947
→1496 : 669947
→1497 : 669947
→1498 : 669947
→1499 : 669947
→1500 : 669947
```

Үр дүн k=380 9 минут:

```
→1474 : 441374
→1475 : 441374
→1476 : 441374
→1477 : 441374
→1478 : 441374
→1479 : 441374
→1480 : 441374
→1481 : 441374
→1482 : 441374
→1483 : 441374
→1484 : 441374
→1485 : 441374
→1486 : 441374
→1487 : 441374
→1488 : 441374
→1489 : 441374
→1490 : 441374
→1491 : 441374
→1492 : 441374
→1493 : 441374
→1494 : 441374
```

# Дүгнэлт

Кодны үр дүнд дүгнэлт хийх нь



# Дүгнэлт

**Table 4 Comparison of the CNDP objective value after removing 10% and 20% of vertices from each network in Table 1**

| Problem | *K* | SEQ | Degree | PageRank | Authority | MIS |
|---|---|---|---|---|---|---|
| Comnat | 2313 | 58,796,393 | 103,398,683 | 87,630,163 | 126,804,602 | NA |
| | 4627 | 83,686 | 90,610 | 92,242 | 7,399,785 | NA |
| Ego | 404 | 2,717,347 | 5,339,614 | 3,816,109 | 6,320,816 | 2,192,636 |
| | 808 | 1,848,740 | 2,070,535 | 2,886,709 | 3,438,031 | 903,441 |
| Flight | 294 | 322,527 | 484,331 | 467,962 | 1,014,305 | 77,777 |
| | 588 | 1,457 | 1,698 | 1,715 | 1,567 | 2,626 |
| Powergrid | 494 | 22,182 | 51,508 | 212,369 | 56,815 | 25,253 |
| | 988 | 3,639 | 4,580 | 14,744 | 3,771 | 5,378 |
| Relativity | 524 | 224,010 | 1,628,337 | 302,309 | 3,382,195 | 23,620 |
| | 1,048 | 4,089 | 4,896 | 9,023 | 6,390 | 6,163 |
| Oclinks | 190 | 637,936 | 785,662 | 758,328 | 835,297 | 746,085 |
| | 380 | 218,215 | 258,277 | 246,876 | 306,289 | 402,824 |

SEQ-based results are those obtained by the proposed algorithm. The MIS-based approach was unable to arrive at solutions within 40 h for the conmat networks.

"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.

```
->387 : 846979
->388 : 846979
->389 : 846979
->390 : 846979
->391 : 846979
->392 : 846979
->393 : 846979
->394 : 846979
->395 : 846979
->396 : 846979
->397 : 846979
->398 : 846979
->399 : 845678
->400 : 845678
->401 : 845678
```

# 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
- articulation_points — NetworkX 3.1 documentation