Introduction

From point of view of this document, the problem is just to maximize the Q function according to a given graph by partitioning the vertices of the graph. Maximizing Q means to minimize inter-group edges meanwhile inner-group edges are maximized. In this text, we are using the simple genetic algorithm to achieve the goal. For the rest of the document chromosome means the following structure:

p: as number of groups		
x_1: the group which v_1 belongs to	$\cdots$	x_n: the group which v_n belongs to

The maximum number of possible groups is equal to the number of vertices, n, therefore we label groups as $0, \ldots, n-1$, which lower group number corresponds to the higher frequent group. If you are asking why the last condition is necessary, it is explained later. After defining the chromosome we modify the main genetic operations, e.g. mutation and etc ..., according to the chromosome and problem structure.

Code Detail

There is some useful information about implementation detail for the mentioned operations. For each operation, three distinct explanation approaches, natural English, Input-Output, Pseducode, are provided. That's enough to understand all the details but you can have a look at the actual code in the given path.

Initiation

To create a random initial chromosome we just assign each vertex to a random group from $0, \ldots, n-1$. After that, we do normalize the Chromosome, which means to reorder group numbers to put most larger groups at the first indices.

The code that is used for this purpose is something like this:

- Input
- Output

  Chromosome $\left(\begin{array}{|c|c|c|} \hline \multicolumn{3}{|c|}{\text{p = number of vertics}} \\ \hline \cdots & x_i = \text{a random number from 0 to } n-1 & \cdots \\ \hline \end{array}\right)$

- Pseudocode

```
function randomChromosome()
  let x be a new chromosome
  for i in 0 ... n do
    x.genome[i] = a random number from 0 ... n - 1
  end
  return nomalized x
end
```

- C++ (you can find this on Solution1/Algorithm.cpp)

```
virtual Chromosome *randomChromosome() const override
{
    Chromosome *chromosome = new Chromosome(problem);
    int problemSize = problem->size();
    chromosome->numberOfGroups = problemSize;
    for(int i = 0; i < problemSize; ++i) {
        (*chromosome)[i] = Random::randomInt(0, problemSize - 1);
    }

    this->_normalize(chromosome);
    return chromosome;
}
```

Mutation

We use mutation to remove small groups and merge them into larger groups. We claim that in the initiation new chromosomes have enough distinct groups. It's easy to show that if the degree of a vertex is zero then we are free to place it in an arbitrary group, it will not affect the value of Q. Therefore there are at most $n/2$ distinct groups in the optimal solution. The following graph shows the number of distinct groups in a chromosome drawn from the above random function averaged over 1000 tests. This shows there are about $6n/10$ distinct groups in a randomly generated chromosome. (we will contemplate this soon.)

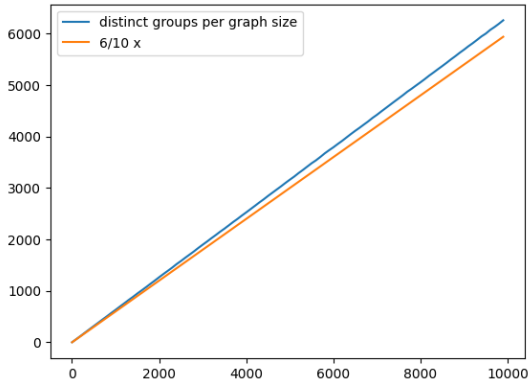
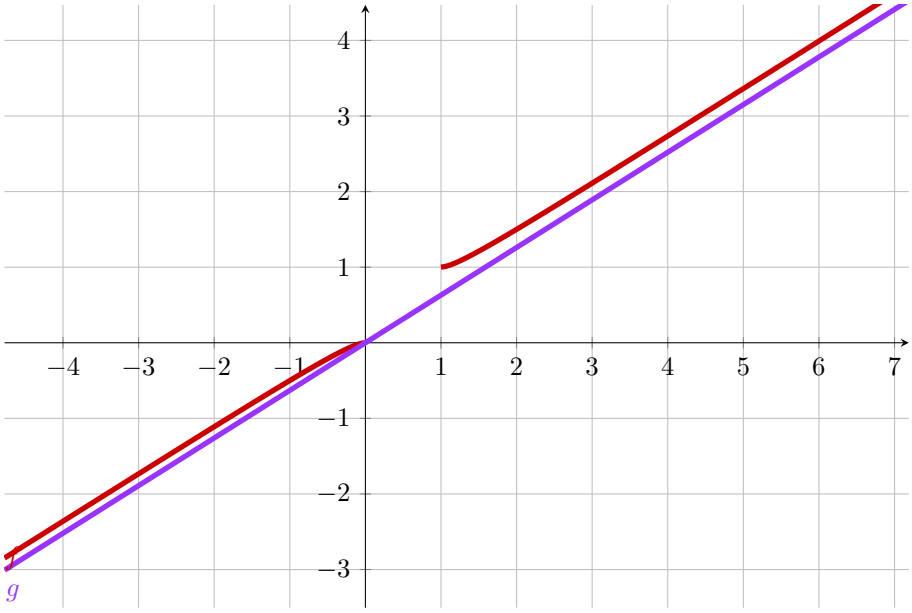


Figure 1: Code which generates this graph is placed in numberOfDistinctGroupsInARandomChromosome.py

It is not hard to proof that expected value of unique values in drawing n numbers from 1 to n with replacement is equal to $n \times (1 - (\frac{n-1}{n})^n)$ which converges to $n \times (1 - \frac{1}{e})$ or approximately $n \times 0.63$. Following plot shows $n \times (1 - \frac{1}{e})$ as red plot and $n \times 0.63$ as purple line. As we can see even in small numbers they are nearly the same.



As you may notice the number of communities in a graph would be way less than half of the graph vertices. Therefore destroying small groups is reasonable and is suitable for the optimality of the solution. In practice, to do so, we select a random group with the probability of reciprocal of its size and assign its members to other groups randomly.

Here we devise an implementation

- Input

Chromosome (| p | | |
|---|---|---|
| x_1 | $\cdots$ | x_n |)

- Output

Chromosome (| p - 1 | | |
|---|---|---|
| ... | if(x_i == a randomly selected group)
then
  a random number from 0 ... p - 1
else
  x_i
end | ... |)

- Pseudocode

```
function selectGroup(groupsSizes as s)
  c = [ 0 ] -- cumulative sum of 1/s for s in grousp sizes s --
  c[i] = c[i - 1] + 1/s[i] -- s is array of groups sizes --

  rc = a random number from [0, last element of c]

  return index of first element of c which is greater than or equal to rc
end

function mutate(chromosome)
  s = selectGroup(chromosome.groupSizes)
  for vertex, group in chromosome do
    if(group == s) then
      adj = select a random neighbour of vertex
      if adj == none:
        group = 0 -- if vertex has no adjacent assign it to an special group --
      else
        group = groups[adj] -- join vertex to the group of one of its adjacents --
      end
    end
  end
end
```

- C++ (you can find this on Solution1/Algorithm.cpp)

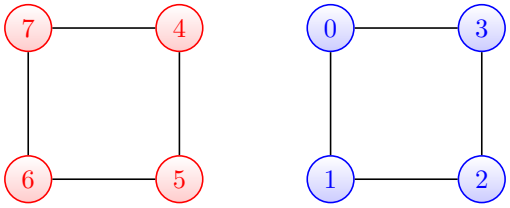
```
virtual Chromosome *Algorithm::mutate(const Chromosome *chromosome) const override;
```

Crossover

In crossover, by definition, we merge two different solution into one in a meaningful way. Here the algorithm is to assign each vertex to the group of corespond vertex in one of its parents randomly. So if we name parents' groups array as Y and Z then X defined as below

$$x_i = P(u > 0) \times (y_i - z_i) + z_i \qquad\qquad u \sim \mathcal{N}(0, \sigma^2)\,.$$

In many cases above function will not produce a good result. For example consider following graph



The communities with best Q are shown with different colors. Now consider following chromosomes

$X =$

p = 2							
0	1	2	3	4	5	6	7
0	0	0	0	0	1	1	1

$Y =$

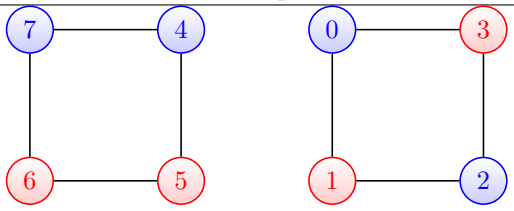
p = 2							
0	1	2	3	4	5	6	7
1	1	1	0	0	0	0	0

These solution are close to optimal solution therefore we expect the child solution after crossover be almost optimal but the result is following chromosome.

Chromosome

p = 2							
0	1	2	3	4	5	6	7
1	0	1	0	0	1	1	0

Graph



And one can be see that in general in this approach result chromosome is random and hardly related to it's parents. This problem is caused by different group numbers, and because of this, we should reorder group numbers based on a property to keep crossover effective. Here we reorder groups according to their vertices. This reordering follows only one rule:

If the smallest vertex of group a is less than that of b then label of a should be less than b.

Throughout this document, this reordering is called normalizing and after each operation, each chromosome undergoes this normalization.

Now after this operation X and Y are as follow.

$X =$

p = 2							
0	1	2	3	4	5	6	7
0	0	0	0	0	1	1	1

$Y =$

p = 2							
0	1	2	3	4	5	6	7
0	0	0	1	1	1	1	1

If we run crossover again the result would be one of the following chromosomes(with same probability), which are way better and more related to the parents than the previous result.

$Z_{00} =$

p = 2							
0	1	2	3	4	5	6	7
0	0	0	0	0	1	1	1

$Z_{01} =$

p = 2							
0	1	2	3	4	5	6	7
0	0	0	0	1	1	1	1

$Z_{10} =$

p = 2							
0	1	2	3	4	5	6	7
0	0	0	1	0	1	1	1

$Z_{11} =$

p = 2							
0	1	2	3	4	5	6	7
0	0	0	1	1	1	1	1

- Input

  X $\left(\begin{array}{|c|c|c|}\hline p & & \\ \hline x_1 & \cdots & x_n \\ \hline \end{array}\right)$

  Y $\left(\begin{array}{|c|c|c|}\hline p' & & \\ \hline y_1 & \cdots & y_n \\ \hline \end{array}\right)$

- Output

  Chromosome $\left(\begin{array}{|c|c|c|}\hline \multicolumn{3}{|c|}{\text{at most } \max\{p, p'\}} \\ \hline \cdots & \text{randomly selected from } x_i \text{ and } y_i & \cdots \\ \hline \end{array}\right)$

- Pseudocode

```
function crossover(x, y)
  let z be a new chormosome

  z.p = max(x.p, y.p)

  for i in 0 ... number of vertices do
    z.genomes[i] = randomInt(0, 1) * (x.genomes[i] - y.genomes[i]) + y.genomes[i]
  end

  return nomalized z
end
```

- C++ (you can find this on Solution1/Algorithm.cpp)

```
virtual Chromosome *crossover(const Chromosome *X, const Chromosome *Y) const override
{
    Chromosome *newChromosome = new Chromosome(problem);
    int problemSize = problem->size();

    newChromosome->numberOfGroups = std::max(X->numberOfGroups, Y->numberOfGroups);

    for(int i = 0; i < problemSize; ++i) {
        if(Random::randomInt(0, 1)) {
            (*newChromosome)[i] = (*X)[i];
        } else {
            (*newChromosome)[i] = (*Y)[i];
        }
    }

    this->_normalize(newChromosome);
    return newChromosome;
}
```

Execution

Build

To build project the you can easily use provided MakeFiles by running `make all` or `make win` for windows in command line, then you can run the code by executing `console.exe` or `console.out` . For compile manually use following command

```
g++ -std=c++20 console.cpp -o console -I.
```

Inputs

The program expects you to put a file name `graph.txt` in the project directory containing graph edges. If file does not exsist program asks for new file name and you can enter `stdin` to enter graph manually.

Outputs

The program generates some useful logs but the final result will be displayed and be written to a file named `communities.py` . Also, a colored graph is generated in dot format which you can use Graphviz to visualize, this graph is written to `graph.dot` . After running the code you can run script `compareResults.py` to compare this code outputs with `networkx` 's output.

An example of output graph is shown below(using provided the dataset.txt)