بسم الله الرحمن الرحیم

عنوان پروژه: پیاده سازی جدول سودوکو 9×9 به وسیله الگوریتم ژنتیک

نام دانشجو: پرنیان نیک پرور

استاد مربوطه: دکتر اکرم بیگی

بهار 1403

فهرست

بخش اول.

*توجه: من یک نمونه جدول سودوکوی حل نشده با عنوان sample.txt در فایل تکست ذخیره کرده ام.

در این قسمت قصد دارم به توضیح فایل GAS.h بپردازم:

در واقع این header فقط تعریف اولیه مقادیر مورد نیاز است و توضیحات بیشتر را در ادامه خواهم داد!

در ابتدا متغیرهایی بصورت const تعریف کردم که کارکرد آن ها مشخص میباشد.

```
const int POPULATION = 400;
const int GENERATION = 100000;
const int CHILDREN = 100;
const int puzzleSize = 9;
const float MUTATION_RATE_MIN = 0.01;
const float MUTATION_RATE_MAX = 0.05;
const float MUTATION_RATE_INCREASE = (MUTATION_RATE_MAX -
MUTATION_RATE_MIN)/GENERATION;
```

سپس در ادامه یک struct Fitness داریم.

در واقع کاربرد این نگهداری چند مورد میباشد:

1. اندیس عنصر
2. والد بودن یا نبودن
3. مقدار fitness هر عنصر

و همچنین 2 اپراتور مقایسه ای که جلوتر در بخش اصلی کد به توضیح آن ها خواهم پرداخت.

```
struct Fitness {
    Fitness(); //Constructor

    int index;
    bool isParent; //If "isParent == 1" means our element is a parent
    float fitness; //Element's fitness

    bool operator<(const Fitness& rhs) const;
    bool operator==(const Fitness& rhs) const;
};
```

در ادامه یک کلاس با نام Candidate داریم که شامل چند تابع و متغیر میباشد.

1. تابع create: این تابع برای ساختن یک آرایه کاندید(که این آرایه خودش یک پازل است) به کار میرود. در واقع این تابع با گرفتن size که اندازه پازل است و همچنین گرفتن آرایه sample که جلوتر به توضیح این آرایه میپردازم یک پازل کاندید برای والد شدن ایجاد میکند.
2. تابع print: این تابع برای چاپ پازل بصورت شکیل و شبیه جدول سودوکو به کار میرود.
3. تابع copy: این تابع یک کاندید دیگر با نام other میسازد و آن را کاندید فعلی کپی میکند.

4. تابع crossover: این تابع وظیفه crossover کردن 2 والد را برای ساخت فرزند دارد که از 3 روش استفاده میکند که جلوتر به آن میپردازیم.
5. تابع mutation: این تابع برای جهش فرزندان به کار میرود.
6. تابع calcFitness: محاسبه fitness که جلوتر به توضیح کارکرد این تابع خواهم پرداخت.

```
class Candidate{
    public :
        void create(int size , int sample[]);
        void print();
        void copy(const Candidate & other);
        void crossover(Candidate & parentA , Candidate & parentB);
        void mutation(int sample[]);
        float calcFitness();
        bool operator<(const Candidate& rhs) const;
        float fitness;
        int puzzle[81];
        int puzzleSize;

};
```

در نهایت یک کلاس سودوکو داریم که پاسخ نهایی، یک شیء از این کلاس میباشد.

این کلاس فایل sample.txt را در یک آرایه ذخیره میکند و همچنین توابع دیگر که در بخش اصلی کد به توضیح آنها خواهم پرداخت.

```
class Sudoku{
    public:
        Sudoku(int nChildren, int nParent); //Constructor
        ~Sudoku();

        void readFromFile(string szFileName);//Read from a file and save in an
array  => (sample[])
        void test();
        void solveElitair(bool isPrint=true);

    private:
        int sample[81];
        //const int puzzleSize = 9; //9
        vector <Candidate> parents, children;
};
```

بخش دوم.

- Main:

در این بخش ابتدا یک جدول سودوکو به وسیله کلاس Sudoku که در هدر فایل تعریف شده بود میسازیم و مقدار ورودی هایش را اندازه CHILDREN و POPULATIONمیدهیم.
سپس برای این شیء، تابع readFromFile را صدا میزنیم.
این تابع که جلوتر به جزئیات کارکرد آن خواهم پرداخت، تمام مقادیری که در sample.txt ذخیره کرده ایم را در آرایه 81 خانه ای []sample ذخیره میکند.
و سپس تابع solveElitair را برای حل سودوکو صدا میزند که در جلوتر به آن خواهم پرداخت.

```
int main(){
    Sudoku sudoko(CHILDREN, POPULATION);
    srand(time(NULL));
        sudoko.readFromFile("sample.txt");
        cout << "done!\n";
        sudoko.solveElitair(true);
    return 0;
}
```

- Fitness:

برای سورت کردن و تعیین یکنواختی fitness ها از دو اپراتور استفاده میکنیم که یک شی از ساختار مربوطه میسازد و با شی فعلی مقایسه میکند و به این ترتیب مرتب سازی انجام میگردد.

```
bool Fitness::operator<(const Fitness& rhs) const {
    return (fitness< rhs.fitness); //Sort Fitness
}


bool Fitness::operator==(const Fitness& rhs) const{
    return (fitness == rhs.fitness); //Determine unifueness
}

//Constructor
Fitness::Fitness(){
    fitness= 0.0;
    isParent = false;
    index = 0;
}
```

- Candidate:
  - ✓ Create():

در حلقه for اولیه، ابتدا تمامی خانه های آرایه ی puzzle را با آرایه sample که با مقادیر فایل sample.txt پر شده اند، پر میکنیم.

0	7	0	0	0	0	0	3	0
3	0	0	0	0	0	0	2	0
0	0	1	0	0	0	0	0	8
0	4	0	0	0	0	0	9	0
8	0	0	0	0	0	0	0	2
0	0	0	3	0	0	0	0	0
0	0	0	0	0	0	3	0	0
0	0	0	0	0	3	0	0	0
0	1	0	0	0	0	0	0	0

سپس داخل حلقه ی for بعدی میرویم که از 0 تا 81، 9 تا 9 تا پیش میرود.(یعنی اول هر سطر)

در داخل حلقه بعدی فرض میکنیم value ما در ابتدا یعنی مقدار 1 می باشد:

در واقع ما الان در خانه 0 از جدول و با Value مقدار 1 هستیم که در اینجا در خط بعد مقدار valueFound=false میشود.

سپس در گام بعدی وارد حلقه ی for میشویم که از 0 تا 9 یعنی تمام خانه های آن سطر را بررسی میکند.

در اینجا مقدار block و square ما هر دو، 0 میباشد.

حال بررسی میکند که مقدار خانه ی puzzle[0] با مقدار value کنونی که 1 است برابر است یا خیر اگر برابر باشد valueFound =true میشود که در اینجا برابر نیست و valueFound = false میماند و وارد شرط میشویم.

در داخل if یک پوزیشن رندوم بین 9 خانه به ما داده میشود.

اگر **فرض کنیم** به ما پوزیشن 7 را بصورت رندوم بدهد، مقدار آن 3 میباشد.

سپس در while میگوید تا زمانی که مقدار آن خانه ی پازل که برای ما 3 == puzzle[7] == puzzle[0+7] میباشد، مخالف 0 بود، مقدار position را 1 واحد افزایش بده و باقی مانده بر تعداد 9 را با position برابر قرار بده که در اینجا برای ما عدد 8 میشود که مقدار آن 0 است، در نتیجه از while خارج شده و value = 1 به این خانه اختصاص داده میشود.

پس puzzle[8] == 1 میگردد.

و به اینصورت تمام خانه های آرایه پازل بین اعداد 1 تا 9 مقدار دهی میشوند و این پازل کاندید ما برای crossover و mutation میگردد.

```
void Candidate::create(int size , int sample[]){
    int value , block , square , position;
    bool valueFound;
    puzzleSize = size;//9

    //Copy all Sample[] in puzzle[];
    for(position = 0 ; position < puzzleSize*puzzleSize; position++){
        puzzle[position]=sample[position];
    }
```

```
    //This loop goes through the entire sudoku, and indexes that have'nt a
value, make their valueFound false
    for(block = 0; block < puzzleSize*puzzleSize; block+= puzzleSize){
        for(value = 1 ; value<= puzzleSize; value++){
            valueFound = false;
            for(square=0; square<puzzleSize; square++){
                if(value==puzzle[block+square]){
                    valueFound = true;
                }
            }
            if(!valueFound){
                    position= rand()%puzzleSize;
                    while(puzzle[block+position]!=0){
                        position = (position+1)%puzzleSize;
                    }
                    puzzle[block+position] = value;
            }
        }
    }
}
```

✓ CalcFitness():

برای محاسبه fitness ما 3 معیار داریم:

Sum، Product و Completeness:

اگر ما در حالت optimal برای جدول سودوکو باشیم میبایست Sum در هر ردیف یا ستون برابر 45 شود.

1+2+3+4+5+6+7+8+9 = 45←

همچنین Product در هر ردیف باید برابر !9 باشد که این مقدار با مقدار حقیقی متفاوت است.

در نهایت ما Completeness را داریم که تعداد اعداد مختلف در هر ردیف یا ستون است.

در واقع در حالت optimal ما 9 عدد متفاوت در هر ردیف یا ستون داریم پس completeness ما برابر 9 میباشد.

در این بخش کد ما sumFitness داریم که برابراست با مجموع اعداد واقعی منهای 45:

$$45\text{-}\Sigma\ value$$

و همچنین complFitness تعداد اعداد مختلف در هر ردیف یا ستون میباشد.

آرایه seen آرایه ایست برای ذخیره تکرار اعداد در هر ردیف یا ستون مثلا اگر 1 دیده شود، seen آن true میشود.

و optsum و optprod هم مجموع و حاصلضرب ایده آل است.

برای محاسبه fitness در هر ردیف یا ستون، مجموع و حاصلضرب اعداد محاسبه شده و تعداد اعداد مختلف در آن ردیف شمرده می‌شودو فاصله مجموع و حاصلضرب واقعی از ایده‌آل و تعداد عدم تکرار اعداد به fitness اضافه می‌شود.

```
float Candidate::calcFitness(){
    int sumFitness = 0 , complFitness = 0;
    int i , j , k , l;
    int nrt = sqrt(puzzleSize); //3  n root
    int sum , prod , completeness;
    int seen[9];
    int optprod = (puzzleSize==4?24:362880); // =n!
    int optsum = (puzzleSize==4?10:45);
    float prodFitness = 0.0;

    //per row
    for(i= 0; i< puzzleSize*puzzleSize; i+= nrt*puzzleSize){
        for(j=0; j<puzzleSize; j+=nrt){
            sum = 0 ;
            prod = 1;
            completeness = 0;

            for(k=0; k< puzzleSize*nrt; k+=puzzleSize){
                for(l=0 ; l<nrt; l++){
                    sum += puzzle[i+j+k+l];
                    prod*=puzzle[i+j+k+l];
                    seen[puzzle[i+j+k+l]-1] = true;   //track occurence of
values
                }
            }

            for(k=0; k<puzzleSize;k++){
                if(seen[k]){
                    completeness++;   //count number of different numbers
                }
            }

            sumFitness += abs(optsum-sum);\
            prodFitness += sqrt(abs(optprod-prod));
            complFitness += puzzleSize-completeness;
        }
    }

    //per column
    for(k=0; k<puzzleSize*nrt; k+=puzzleSize){
        for(l=0; l<nrt; l++){
            sum = 0;
            prod =1 ;
            completeness = 0;

            for(i=0; i<puzzleSize*puzzleSize; i+= nrt*puzzleSize){
                for(j=0; j<puzzleSize; j+=nrt){
                    sum += puzzle[i+j+k+l];
```

```
                    prod *= puzzle[i+j+k+l];
                    seen[puzzle[i+j+k+l]-1] = true;
                }
            }
            for(i=0 ; i<puzzleSize; i++){
                if(seen[i]){
                    completeness++;
                }
            }
            sumFitness += abs(optsum-sum);
            prodFitness += sqrt(abs(optprod-prod));
            complFitness += puzzleSize-completeness;
        }
    }

    fitness = 10.0*sumFitness + prodFitness + 50.0*complFitness;
    return fitness;
}
//Swap the value of two random position in a random block
void Candidate::mutation(int sample[]){
    int block = rand()%puzzleSize;
    int positionA = rand()%puzzleSize;
    int positionB = rand()%puzzleSize;
    int temp;
    int  nAttempts = 0;
    while((positionA==positionB || sample[block*puzzleSize+positionA] ||
sample[block*puzzleSize+positionB]) && nAttempts<puzzleSize*puzzleSize){
        block = rand()%puzzleSize;
        positionA = rand()%puzzleSize;
        positionB = rand()%puzzleSize;
        nAttempts++;
    }

    if(nAttempts>=puzzleSize*puzzleSize){
        //finish
        return;
    }
    temp = puzzle[block*puzzleSize+positionA];
    puzzle[block*puzzleSize+positionA] = puzzle[block*puzzleSize+positionB];
    puzzle[block*puzzleSize+positionB] = temp;
}
```

✓ Mutation():

برای جهش، ابتدا یک بلوک و 2 پوزیشن رندوم کمتر از 9 انتخاب میکنیم.

حالا اگر(این 2 پوزیشن هر دو یک خانه باشند یا یکی از این پوزیشن ها از given های قطعی فایل sample باشد که نباید تغییر کنند) و علاوه بر آن مقدار nAtempts<81 باشد:

دوباره بصورت رندوم بلوک و 2 پوزیشن را انتخاب میکند تا زمانی که این شرط While برقرار نباشد.

اگر هم nAttempts>=81 شود یعنی به انتهای پازل رسیدیم و تمام شده است.

حال اگر هیچ یک از اتفاقات بالا برقرار نبود جای 2 پوزیشن را عوض میکنیم.

```
void Candidate::mutation(int sample[]){
    int block = rand()%puzzleSize;
    int positionA = rand()%puzzleSize;
    int positionB = rand()%puzzleSize;
    int temp;
    int  nAttempts = 0;
    while((positionA==positionB || sample[block*puzzleSize+positionA] ||
sample[block*puzzleSize+positionB]) && nAttempts<puzzleSize*puzzleSize){
        block = rand()%puzzleSize;
        positionA = rand()%puzzleSize;
        positionB = rand()%puzzleSize;
        nAttempts++;
    }

    if(nAttempts>=puzzleSize*puzzleSize){
        //finish
        return;
    }
    temp = puzzle[block*puzzleSize+positionA];
    puzzle[block*puzzleSize+positionA] = puzzle[block*puzzleSize+positionB];
    puzzle[block*puzzleSize+positionB] = temp;
}
```

✓ Crossover():

ورودی تابع crossover دو شیء از کلاس Candidate است که درواقع ما می آییم دو شیء parentA و parentB را به وسیله Candidate همونطور که بالا توضیح داده شد، میسازیم.

برای انجام crossover سه روش داریم که انتخاب بین این سه روش به صورت رندوم در خط 162 توسط method انتخاب میگردد که به وسیله سوئیچ کیس سه حالت را بررسی میکنیم.

در کیس 0 حلقه for داریم که از 0 تا 81، 9 تا 9 تا پیش میرود.(ردیفی)

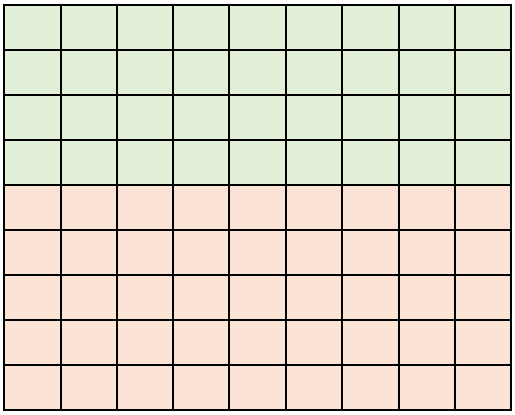
سپس parent بصورت رندوم بین 0 یا 1 انتخاب میگردد.

حال مثلا در ردیف 2، ستون 0 بررسی میکند اگر مقدار parent=1 بود، مقدار خانه i+j مربوط به پازل parentA که یک کاندید بود را در پازل فعلی میریزد و همچنین برای parent=0.

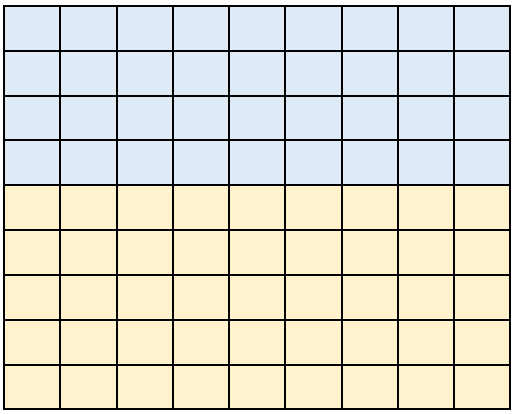
واین کار را برای تمام خانه ها انجام میدهد و به اینصورت بصورت رندوم خانه ها تقطیع میشوند.

در کیس 1 بصورت ردیفی و در 2 بصورت ستونی مطابق شکل تقطیع رخ میدهد:

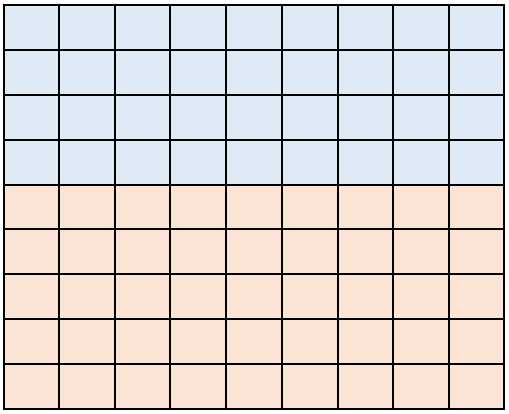
parentA:



parentB:



Crossoverd child:



و به همین صورت برای ستون ها:

```
void Candidate::crossover(Candidate & parentA, Candidate & parentB){
    int i , j , nrt;
    int method = rand()%3;
    int offset;
    bool parent;
    puzzleSize = parentA.puzzleSize;
    nrt = sqrt(puzzleSize);

    switch(method){
        case 0 : //Random
            for(i=0; i<puzzleSize*puzzleSize; i+=puzzleSize){
                parent = rand()%2;
                for(j=0 ; j< puzzleSize; j++){
                    if(parent){
                        puzzle[i+j] = parentA.puzzle[i+j];
                    }
                    else{
```

```
                        puzzle[i+j] = parentB.puzzle[i+j];
                    }
                }
            }
        break;

        case 1: //per Row
            offset = 1+rand()%(nrt-1);
            for(i=0; i< offset*puzzleSize*nrt; i++){
                puzzle[i] = parentA.puzzle[i];
            }
        break;

        case 2: //per Column
            offset = 1+rand()%(nrt-1);
            for(i=0; i<puzzleSize*puzzleSize; i+=puzzleSize){
                for(j=0; j<offset*puzzleSize; j++){
                    puzzle[i+j] = parentB.puzzle[i+j];
                }
            }
        break;

        default:
            cout<< "No Crossover. \n";
        break;
    }
}
```

✓ Print():

این تابع نکته خاصی ندارد فقط برای چاپ شدن با فرم مرتب پاسخ سودوکو میباشد.

```
void Candidate::print(){
    int i, j , k , l, nrt = sqrt(puzzleSize);

    for(i=0; i<puzzleSize*puzzleSize; i+=nrt*puzzleSize){
        for(j=0 ; j<puzzleSize; j+=nrt){
            for(k=0; k<puzzleSize*nrt; k+=puzzleSize){
                for(l=0; l<nrt; l++){
                    cout << puzzle[i+j+k+l] << " ";
                }
                if(k<(puzzleSize*nrt)-puzzleSize){
                    cout << "| ";
                }
            }
            cout << endl;
        }
        if(i< (puzzleSize*puzzleSize)-(nrt*puzzleSize)){
```

```
            for(j=0; j<puzzleSize+nrt-1;j++){
                cout << "- ";
            }
            cout << endl;
        }
    }
    cout << endl;
}
```

✓ Copy():

این تابع یک کاندید به نام other میسازد و پازل آن را در پازل فعلی میریزد.

```
void Candidate::copy(const Candidate & other){
    int i;

    puzzleSize = other.puzzleSize;
    for(i=0; i<puzzleSize*puzzleSize; i++){
        puzzle[i] = other.puzzle[i];
    }
}
```

- Sudoku():

✓ Soduko():

در این بخش یک جدول سودوکو با تعدادی پرنت و چیلدرن ساخته میشود که کاندید هایی در این پرنت و چیلدرن push میشوند که این کار داخل مین سبب ایجاد Sudoku sudoku شد!

```
Sudoku::Sudoku(int nChildren, int nParents){
    int i;
    for(i=0; i< nChildren; i++){
        children.push_back(Candidate());
    }
    for(i=0; i<nParents; i++){
        parents.push_back(Candidate());
    }
}
```

✓ ~Sudoku():

حافظه پرنت و چیلدرن را خالی میکنیم.

```
Sudoku::~Sudoku(){
    children.clear();
    parents.clear();
}
```

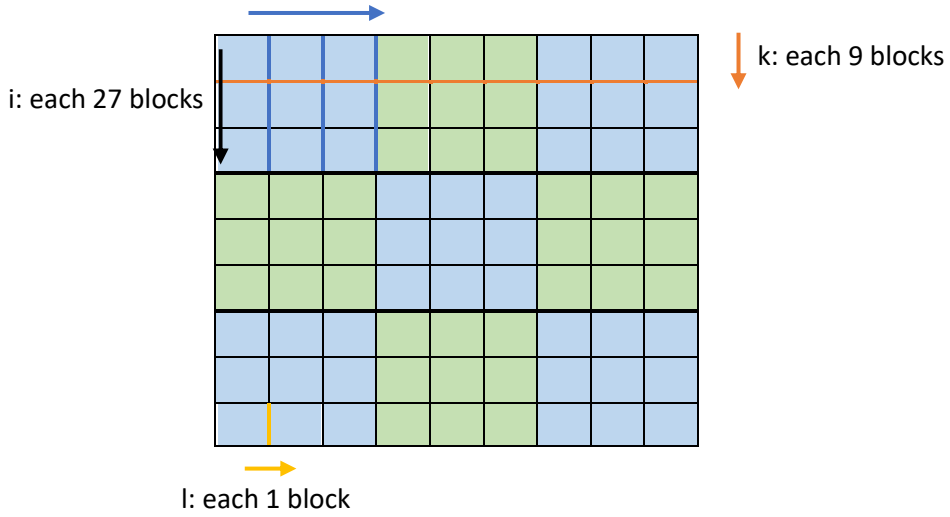
✓ readFromFile():

جدول سودوکو در کل متشکل از 81 بلوک است که هر مربع 3×3 آن را یک باکس در نظر میگیریم و در مجموع 9 باکس داریم.

در کد این بخش ما یک حلقه for برای i داریم که مقدار آن به اندازه puzzleSize*nrt یا بعبارتی 3*9 هربار بیشتر میشود. این پیمایش بیانگر هر سطر باکسی ماست. در واقع وقتی باکس های ردیف اول تمام شد به دسته بندی باکس های بعدی میرویم.

درباره حلقه j باید گفت که این حلقه هربار به اندازه nrt افزایش میابد یا بعبارتی هربار 3 خانه جلو میرود که این هم نشان دهنده هر باکس در ستون میباشد.

درباره حلقه k هم داریم، این حلقه در سطر هر باکس پیش میروند و به اندازه puzzleSize یا همان 9 تا هربار به مقدارش افزوده میگردد.




```
void Sudoku::readFromFile(string szFileName){
    fstream file(szFileName.c_str(), ios::in);
    int i , j , k , l;
    int nrt;
    if (!file.is_open()){
        cout << "File not found!\n";
        exit(0);
    }
    nrt = sqrt(puzzleSize);  //3
    //puzzleSize : 9  //nrt*puzzleSize => 27
    //for(i) => each 3 column or we can say each 3×3 sub puzzle(Vertical)
    //for(j) => each 3 block per row or we can say each 3×3 sub
puzzle(Horizontal)
    //for(k) => each row
    //for(l)=> each column
    for(i=0 ; i< puzzleSize*puzzleSize; i+=nrt*puzzleSize){
        for(j=0; j<puzzleSize; j+=nrt){
            for(k=0 ; k< puzzleSize*nrt; k+=puzzleSize){
```

```
                for(l= 0 ; l < nrt; l++){
                    file >> sample[i+j+k+l];
                }
            }
        }
    }
    file.close();
}
```

✓ solveElaiter():

در این تابع، مراحل زیر انجام می‌شود:

ابتدا از parents جمعیت اولیه ایجاد میگردد و fitness آن‌ه محاسبه میشود.

سپس fitParent و fitChildren برای ذخیره fitness پرنت ها و چیلدرن ها و اندیس ها مقداردهی میشوند.

در حلقه while، چیلدرن ها با crossover بین والدها ، و همچنین mutation با احتمال MUTATION_RATE، رخ میدهد و fitness آنها محاسبه میشود. سپس با مرتب‌سازی fitnessبهترین والدین و فرزندان انتخاب میشوند.

سپس بهترین والدین و فرزندان به نسل بعد انتقال میابند. و fitness پنج حالت بهترین در هر نسل پرینت میشود.

در نهایت این تابع درصورت پیدا شدن راه حل optimal یا به پایان رسیدن تعداد Generation ها متوقف میشود و حالت بهینه حل شده و محاسبه fitness آن پرینت میشود.

```
void Sudoku::solveElitair(bool isPrint){
    int i,j , parentA , parentB , generation =0 , nParentsNextGen =
0.25*POPULATION;
    float mutate , MUTATION_RATE = MUTATION_RATE_MIN;

    Candidate parentCopy[POPULATION];
    bool isDone = false;

    vector <Fitness> ::iterator endUnique;
    vector <Fitness>::iterator it;
    vector <Fitness> fitParents;
    vector <Fitness> fitChildren;

    //initialise parents; n random parents, givens at the right place

    for(i=0; i<POPULATION; i++){
        parents[i].create(puzzleSize, sample);
        parents[i].calcFitness();
    }

    //initialise fitness place holder
    fitParents.reserve(POPULATION);
    for(i=0; i<POPULATION; i++){
        fitParents.push_back(Fitness());
    }
```

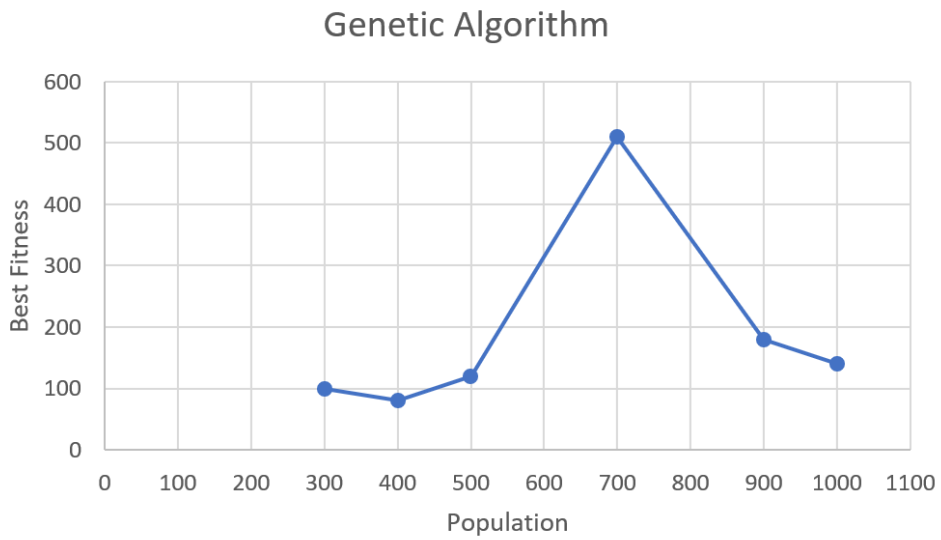
```
fitChildren.reserve(POPULATION);
for(i=0; i<CHILDREN; i++){
    fitChildren.push_back(Fitness());
}

while(generation<GENERATION && !isDone){
    generation++;

    //copy parent fitness to vector

    for(i=0; i<POPULATION; i++){
        fitParents[i].fitness = parents[i].fitness;
        fitParents[i].index = i;
    }

    //Create Children
    for(i=0; i<CHILDREN; i++){
        do{
            parentA = rand()%POPULATION;
            parentB = rand()%POPULATION;
        }while(parentA==parentB);
        mutate = float(rand()%65) /100.0; //will we mutate?

        children[i].crossover(parents[parentA] , parents[parentB]);

        if(mutate<=MUTATION_RATE){
                children[i].mutation(sample);
        }

        fitChildren[i].fitness = children[i].calcFitness();
        fitChildren[i].index = i;
    }

    sort(fitParents.begin(), fitParents.end());
    sort(fitChildren.begin(), fitChildren.end());

    //copying the original parents
    for(i=0; i<POPULATION; i++){
        parentCopy[i] = parents[i];
    }

    //copy n best parents
    endUnique = unique(fitParents.begin() , fitParents.end());
    it = fitParents.begin();
    for(i=0; i<nParentsNextGen; i++){
        parents[i] = parentCopy[it->index];
        if(it != endUnique){
```

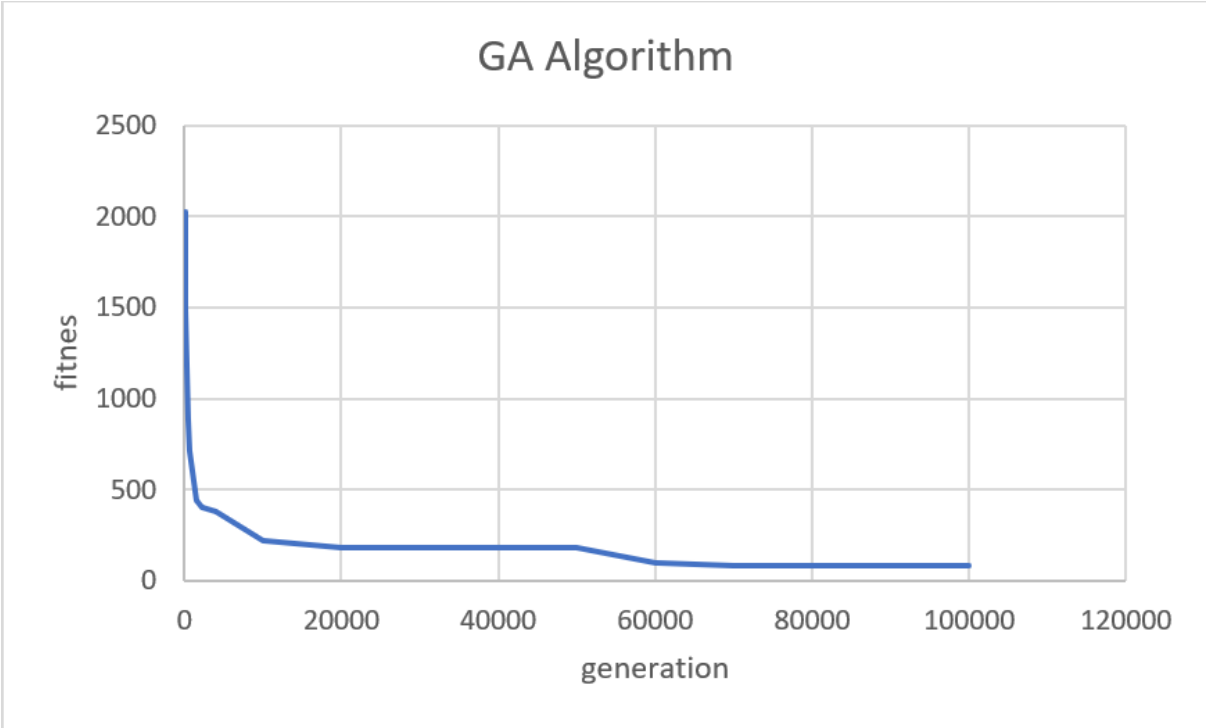
```
                ++it;
            }
            else{
                it=fitParents.begin();
            }
        }

        //copy N best children
        endUnique = unique(fitChildren.begin() , fitChildren.end());
        it = fitChildren.begin();
        for(i=0; i<POPULATION - nParentsNextGen; i++){
            parents[nParentsNextGen+i] = children[it -> index];
            if(it != endUnique){
                ++it;
            }
            else{
                it = fitChildren.begin();
            }
        }

        sort(parents.begin() , parents.end());

        if(isPrint){
            cout << "generation" << generation << ": ";
            for(i=0; i<5 ; i++){
                cout<< parents[i].fitness << " ";
            }
            cout << endl;
        }

        //Stop if optimall solution is reached
        if(parents[0].fitness ==0){
            isDone = true;
        }

        //print boards while running
        if(isPrint && generation%1000==0){
            for(i=0; i<3; i++){
                parents[i].print();
            }
        }
        MUTATION_RATE += MUTATION_RATE_INCREASE;
    }
 if(isPrint){
        parents[0].print();
    }
    cout << parents[0].calcFitness() << endl << generation << endl;
}
```

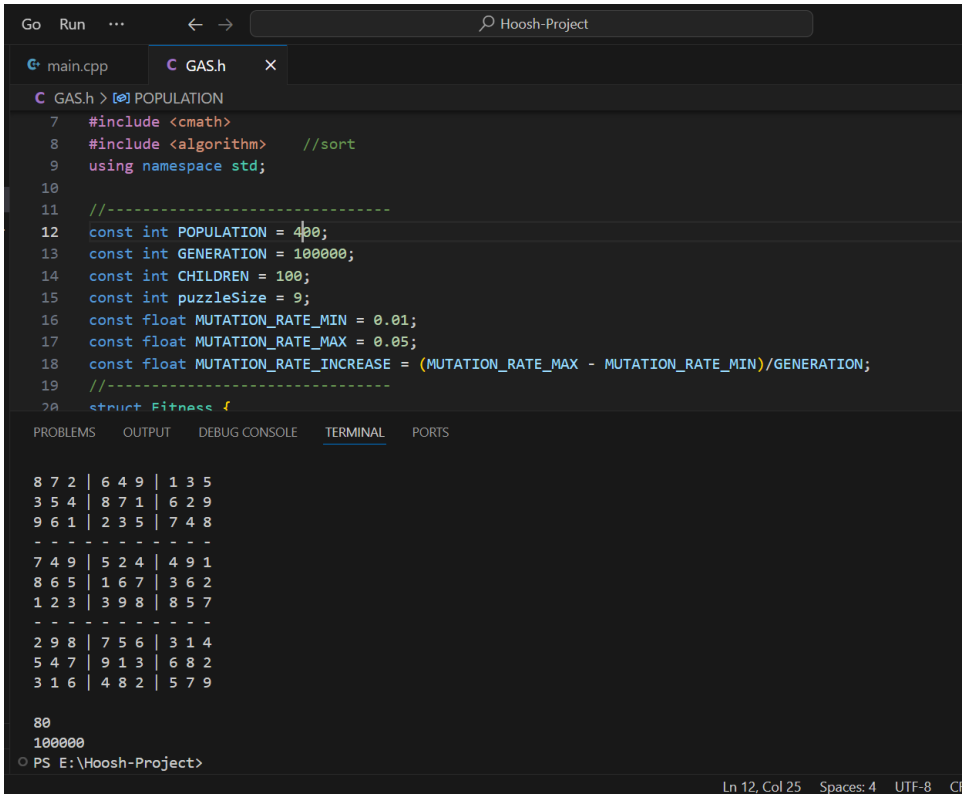
بخش سوم.

نمودار ها:

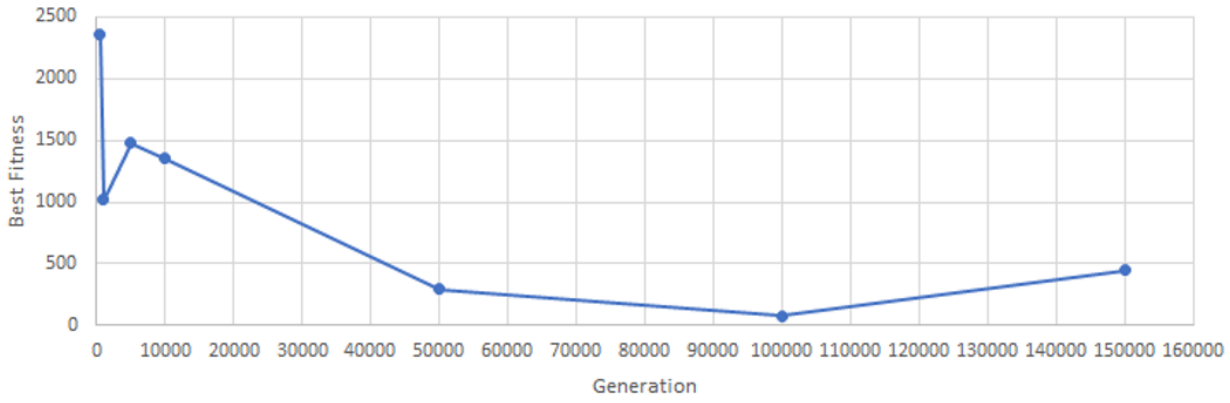
درصورتی که Generation = 100000 و Mutation_Rate_max = 5% باشد، با تغییر Population از 200 تا 1000 مقدار best fitness را اندازه میگیریم که best fitness در population=400 رخ میدهد:



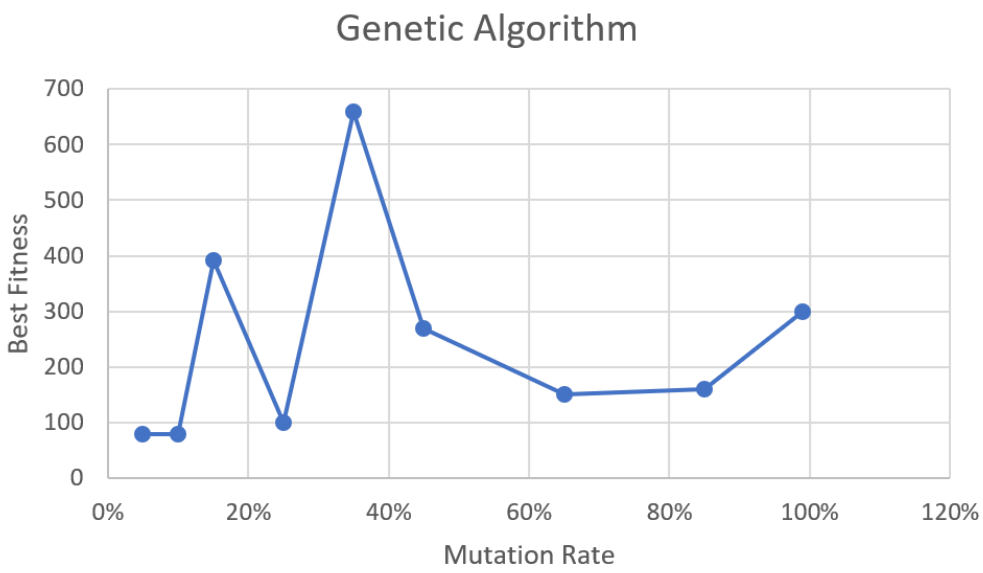




درصورتی که Population = 400 و Mutation_rate_Max = 5% باشد با تغییر Generation از 500 تا 200000 داریم:



در گام بعدی، Population=400 و Generation=100000 را ثابت نگه میداریم و Mutation_Rate_Min را از ٪1 تا ٪45 و Mutation_Rate_Max را از ٪5 تا ٪50 افزایش میدهیم:


Genetic Algorithm
Best Fitness
700
600
500
400
300
200
100
0
0%
20%
40%
60%
80%
100%
120%
Mutation Rate