Assignment 5

Yaniv Raizner 300085677

Artyom Hvedjuk 317572204

General overview

We have implemented our entire genetic engine from scratch for this assignment , because we had problems with the old one and we didn’t get feasible results from it. We have implemented tournament as selection method because we have used other selection algorithms in past such as SUS , PWS and all this algorithms weren’t good as tournament , so we decided to switch to tournament .The mating method that was chosen is uniform , we take genes from each one of the parents with the same probability . The mutation we have chose is taking a graph , then finding a color that appears minimum number of times and switching it with a color that appears maximum times , this way reduce the number of colors by one , though creating a lot of edge collisions.

In order for the program to run we have to specify number of vertices as N variable and then we create a adjacency matrix that holds all the edges, we get the edges from the file specified, format of file should be as follows :

e NUM\_OF\_EDGE NUM\_OF\_EDGE

For Example:

e 1 267

e 1 303

e 1 169

and so on .

At this point we started to acknowledge that using the a adjacency matrix will result in very bad time complexity when iterating over the edges, so we deiced to implement the edges as list<pair<int, int>> edgeslist, we keep a list of pairs in order to iterate on them and not on the adjacency matrix . We also keep track of all the densities for each vertex, so we will use this number later in order to generate our population.

We have used shared pointer a lot in our solution in order not to pass variables by value , we wanted to improve time and space complexity of the program

When we initialize our population we represent each citizen with array of size of N ( number of vertices) , we save a color as number for each vertex . When we initialize this array for each citizen ,we use the max density number we have computed before. We randomly assign colors to each vertex , the maximum number of colors is the maximum density number , it is the upper bound.

**Code for citizen initialization :**

void init\_population(ga\_vector &population,

ga\_vector &buffer )

{

int tsize = GA\_TARGET.size();

for (int i=0; i<GA\_POPSIZE; i++) {

ga\_struct citizen;

citizen.fitness = 0;

citizen.str.erase();

shared\_ptr<array<size\_t,N>> colors = Graph::createRandomColors(maxDensityEdge);

citizen.graph = new Graph(givenGraph,\*colors,N, edgeslist);

population.push\_back(citizen);

buffer.push\_back(citizen);

}

}

**Code for random colors assignment :**

static shared\_ptr<array<size\_t, N>> createRandomColors(int maxColors){

shared\_ptr<array<size\_t, N>> temp(new array<size\_t, N>());

for (int i = 0; i < N; i++)

{

(\*temp)[i] = rand() % maxColors;

}

return temp;

}

The fitness calculating function we used was , if citizen got no conflict vertices ( collisions) we give it a fitness of 0 , if citizen got conflicts we compute it’s fitness with following formula : Kcolor \* number\_of\_conflict\_vertices \* 7 , that way we give weight according to the number of colors used for each citizen. That way we see citizen that solved the coloring problem for some color.

**Code for finding number of conflict vertices :**

shared\_ptr<list<pair<size\_t, size\_t>>> Graph::findAllConflictVertecies(array<size\_t, N> \*input\_colors){

shared\_ptr<list<pair<size\_t, size\_t>>> confictsVertices(new list<pair<size\_t, size\_t>>());

array<size\_t, N> \*temp\_colors;

if (input\_colors != nullptr)

{

temp\_colors = input\_colors;

}

else {

temp\_colors = &p\_colors;

}

int conflictsNumbers[N] = {0};

list<pair<int, int>>::iterator it = p\_edges.begin();

for (; it != p\_edges.end(); ++it)

{

if ((\*temp\_colors)[it->first] == (\*temp\_colors)[it->second])

{

++((conflictsNumbers)[( densityVerticesNumber[it->first] < densityVerticesNumber[it->second] ) ? it->first : it->second ]);

}

}

for (int i = 0; i < N; i++)

{

if (conflictsNumbers[i] > 0)

{

shared\_ptr<pair<size\_t, size\_t>> newPair( new pair<size\_t,size\_t>(i, conflictsNumbers[i]));

(\*confictsVertices).push\_back(\*newPair);

}

}

return confictsVertices;

}

We also implemented a stop function which checks whether we achieved 2 coloring and the fitness of this solution is 0 , that is the minimum k coloring solution.

bool Graph::doWeWantToStop(){

if ((kColor <= 2) && CalcFitness() == 0)

{

return true;

}

return false;

}

Memetics :

We implemented an exploration stage to the genetic engine , the exploration stage consists of 4 different search options , these searches try to solve the coloring problem with given k number or try to reduce the number of K colors and try to solve it with the new K. In order to know what vertices are the “problematic” ones we have a function called findAllConflictVertecies() , this function returns all the color collisions in the graph and the we take care of them. We expand the search tree until certain depth , specified in global variable called const size\_t depth = 2;

For local search to be implemented we first need to define a citizen’s neighbors. We did it basically by taking a citizen and finding all of its conflicts, then for each conflict (which basically means vertex in x color) we have changed its color to be not as one of its adjacent vertices. If we couldn’t find any color to change to, we just randomly chosen a new color. For each change we have created a new citizen’s neighbor, and eventually we had list of citizen’s neighbors.

Then we chose the neighbor to expand according to:

1. Hill climbing : we implemented the hill climbing search , which looks for the best configuration available (by fitness) and goes to it .
2. Tabu search – with this search we are looking for a better solution for the coloring problem we save the steps we did in a list<array<size\_t,N>> it’s a an array of previous configurations , we save them in order not go back to steps we already have taken , we do it in order not to go forth and back to the same solution .
3. Simulated annealing – we have used a variable T to represent a temperature for this search , we are using it to determine whether we want to take a “bad” path hoping it will lead us to better solution .This search is the answer to the problem of hill climbing search which can lead us to local maximum .
4. Mixed mode – we randomly choose a search option for each step , for each step any of the previous searches maybe be applied with same probability . We also tried to implement this function with input that will control how many steps were taken using each search. We could pass 3 percentages which sum to 100% , for example ( 50%,30%,20%) , meaning we would use hill climbing for 50% of citizens , tabu search for 30% and simulated annealing for the rest.

We also implemented an information propagation, from 1 citizen to another. The information we choose to pass on is, when 1 citizen have solved a coloring for certain number we want all the other citizens to stop and try to solve coloring problem with more colors than K , because we already have a solution for this number. We change the colors of all the citizens which still have more than K colors to K-1 colors and let them try to solve this problem.

We also tried to mix between number of citizens who use memetics and other that used only genetic algorithms. To check it’s influence on the solutions ,read the conclusions section .

Summary;

We have ran several kind of configuration, in each one of them we tried to learn something about the genetic algorithm properties influences on the program results together with the new memetic learning exploration.

All this graphs represents data collected when running the algorithm until we got our solution to get to kColor = 7. As lower number of the iteration the better results are.

We started our analysis section examining the local search depth of the exploration phase.

As we can, as expected, ad deeper the depth as better the results. This one was very intuitive for us since we expected this results because increasing the depth is equivalent with running the program for more iteration. The pitfall of bigger depth is slower execution time.

We were very surprise to see the above results, not only that the mutating is not helping to resolve the problem it makes it worse! It destroys the exploration phase… and why? We don’t really have a good explanation, but our best guess is that our mutation function is too destructive.

While we increased the number of the citizens we got better results, until we achieved some kind a of a pitch point in which we couldn’t get better results from there. As we can see, bigger population than 1500 citizens doesn’t improve the results significantly. Moreover, even if does improve the results somehow, it’s not worth the time it takes. And why? Well, let say our algorithm can get to X color as result to given graph, in this case – no matter how many iteration / citizens we would use – we couldn’t improve the results.

We couldn’t get any good conclusion from changing the elitism rate.

\*\* When we moved to check different graphs with different number of vertices, we naturally got different results for each configuration check.

\*\* we used 1000 population size

The first attempt was with very small graph, only 10 vertices, we got immediate results after 2 iteration. The main goal of this test was to print out and check the results in order to validate our algorithm correctness

Second attempt was to compare results with regular exploration and exploration with information propagation. We have stopped the program run when we got to kColor = 7. As we can see the information propagation is awesome stuff, it improved the algorithm iteration number significantly.

Now, as we increase the number of vertices we now don’t stop on certain kColor, instead we run for 50 iteration and check the results. As we can the graph is number of iteration for kColor and the big winner is the Tabu Search with kColor = 13, while others couldn’t get there.

* This ran with no information propagation.

As we moved to 40 vertices graph we have checked once again the regular memtics vs information propagation and the results amazed us, the Tabu Search with information propagation got for as a kColor = 14 which is a lot better.

So we found ourselves trying to understand what so good with Tabu Search and information propagation?

So Tabu Search, all other local search we have implemented can go forth and back from same configuration to same configuration, especially simulating annealing which can go from bad configuration back to previous bad configuration.

And why information propagation give us better results? Since it makes many citizens computations’ redundant to be demolished. In other words, if we have found a solution of let say kColor = 9, there is no reason for no other citizen to try and solve a configuration of kColor >= 9. This frees a lot of citizens to solve relevant configuration problem.

Big graphs,

We have ran many big graphs (from the link in the assignment documentation), it took a lot time to run it even on small populations (~20). And from one example of a graph with 450 vertices ~9000 edged with best known color of 26, we couldn’t get there, out solution got to kColor = 64 after few thousands of iterations.

