cs380su21-meta.sg

Dashboard / My courses / cs380su21-meta.sg / 26 July - 1 August / Assignment 12 (Genetic Algorithm)

Description Submission </> Edit Submission view

Grade

Reviewed on Thursday, 5 August 2021, 3:43 AM by Automatic grade
grade: 100.00 / 100.00

Assessment report [-]
[+]**Summary of tests**

Submitted on Thursday, 5 August 2021, 3:43 AM (Download)

functions.cpp

```
/*!******************************************************************************
\file functions.cpp
\author Vadim Surov, Goh Wei Zhe
\par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu
\par Course: CS380
\par Section: B
\par Programming Assignment 12
\date 4-8-2021
\brief
This file has declarations and definitions that are required for submission
*******************************************************************************/
#include "functions.h"

namespace AI
{


} // end namespace
```

functions.h

```
/*!*****************************************************************************
\file functions.h
\author Vadim Surov, Goh Wei Zhe
\par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu
\par Course: CS380
\par Section: B
\par Programming Assignment 12
\date 4-8-2021
\brief
This file has declarations and definitions that are required for submission
*******************************************************************************/
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <iostream>
#include <sstream>
#include <vector>
#include <string>
#include <cmath>

#include "data.h"

#define UNUSED(x) (void)x;

template<typename Gene>
struct Fitness_Accumulate
{
    /***************************************************************************
    Calculates the sum of all genes in a chromosome

    \param genes
    A vector of genes of type Gene

    \return
    Returns the sum of all genes in a chromosome
    ***************************************************************************/
    int operator()(const std::vector<Gene>& genes) const
    {
        int s = 0;

        for (auto g : genes)
        {
           s += g.getValue();
        }

        return s;
    }
};

template<typename Gene>
struct Fitness_Nbits
{
    /***************************************************************************
    \brief
    Calculates a percentage that indicates the fitness of a particular
    chromosome into a particular solution. Fittest chromosome has all genes
    equal to 1.

    \param genes
    A vector of genes of type Gene

    \return
    Returns the percentage.
    ***************************************************************************/
    int operator()(const std::vector<Gene>& genes) const
    {

        if (genes.size())
        {
            int s = 0;

            for (auto g : genes)
                s += g.getValue();

            return (100 * s) / genes.size();
        }

        return 0;
    }
};

template<typename Gene>
struct Fitness_8queens
{
    /***************************************************************************
    \brief
    Calculates a measure in percentage that indicates the fitness of a
    particular chromosome into a particular solution of 8 queens problem.

    \param genes
    A vector of genes of type Gene

    \return
    Returns the percentage.
    ***************************************************************************/
    int operator()(const std::vector<Gene>& genes) const
    {
        int size = genes.size();
        int max = (size - 1) * size / 2;

        //Check horizontal threats
        int counter = 0;

        for (int j = 0; j < size - 1; ++j)
        {
            for (int i = j + 1; i < size; ++i)
            {
                int dy = genes[j].getValue() - genes[i].getValue();

                if (dy == 0)
                    counter++;
            }
        }

        //Check diagronal threats
        for (int j = 0; j < size - 1; ++j)
        {
            for (int i = j + 1; i < size; ++i)
            {
                int dx = i - j;
                int dy = std::abs(genes[j].getValue() - genes[i].getValue());

                if (dy == dx)
                    counter++;
            }
        }

        return std::abs(100*(max - counter) / max);
    }
};

namespace AI
{
    // Crossover methods for the genetic algorithm
    enum CrossoverMethod { Middle, Random  };

    // Simplest gene seeding class/function
    struct Seed
    {
        int operator()(int p = 0) const
        {
            return p;
        }
    };

    // Gene seeding class/function with a fixed value
    template<int Val = 0>
    struct Seed_Value
    {
        int operator()(int /* p */ = 0) const
        {
            return Val;
        }
    };

    template<int Max>
    struct Seed_Random
    {
        /***********************************************************************
        \brief
        Gene random seeding function

        \return
        Returns random seed value
        ***********************************************************************/
        int operator()(int /* p */ = 0) const
        {
            return static_cast<int>(std::floor(std::rand() % Max));
        }
    };

    // Gene class
    template<typename T = int, typename S = Seed>
    class Gene
    {
        T value;

    public:
        Gene(int p = 0): value{ S()(p) }{}

        T getValue() const
        {
            return value;
        }

        void setValue(T v)
        {
            value = v;
        }

        friend std::ostream& operator<<(std::ostream& os, const Gene& rhs)
        {
            os << rhs.value;
            return os;
        }
    };

    // Chromosome class
    template<typename Gene, typename Fitness, size_t Size>
    class Chromosome
    {
        std::vector<Gene> genes;
        int fitness;

    public:

        using gene_type = Gene;

        static const size_t size = Size;

        Chromosome()
            : genes(Size), fitness{ Fitness()(genes) }{}

        std::vector<Gene>& getGenes()
        {
            return genes;
            return genes;
        }

        void setGenes(const std::vector<Gene>& v)
        {
            genes = v;
            fitness = Fitness()(genes);
        }

        Gene getGene(size_t i) const
        {
            return genes[i];
        }

        void setGene(size_t i, const Gene& v)
        {
            genes[i] = v;
            fitness = Fitness()(genes);
        }

        int getFitness() const
        {
            return fitness;
        }

        // Select a random mutation point and change
        // gene at the mutation point
        void randomMutation()
        {
            setGene(std::rand() % Chromosome::size, Gene());
        }

        // Copy genes from a source
        void copyGenesFrom(Chromosome& src)
        {
            std::copy(src.genes.begin(), src.genes.end(), genes.begin());
            fitness = Fitness()(genes);
        }

        friend std::ostream& operator<<(std::ostream& os,
            const Chromosome& rhs)
        {
            os << '[';
            for (auto it = rhs.genes.begin(); it != rhs.genes.end(); ++it)
                os << *it << (it + 1 != rhs.genes.end() ? "," : "");
            os << "]=" << rhs.fitness;
            return os;
        }
    };

    // Individual class
    template<typename Chromosome>
    class Individual
    {
        Chromosome chromosome;

    public:

        using chromosome_type = Chromosome;
        using gene_type = typename Chromosome::gene_type;

        Individual()
            : chromosome{ }{}

        Chromosome& getChromosome()
        {
            return chromosome;
        }

        std::vector<gene_type>& getGenes()
        {
            return chromosome.getGenes();
        }

        void setGenes(const std::vector<gene_type>& v)
        {
            chromosome.setGenes(v);
        }

        gene_type getGene(size_t i) const
        {
            return chromosome.getGene(i);
        }

        void copyGenesFrom(Individual& individual)
        {
            chromosome.copyGenesFrom(individual.chromosome);
        }

        void setGene(size_t i, gene_type gene)
        {
            chromosome.setGene(i, gene);
        }

        int getFitness() const
        {
            return chromosome.getFitness();
        }

        friend std::ostream& operator<<(std::ostream& os, Individual& rhs)
        {
            os << rhs.chromosome;
            return os;
        }
    };

    // Population class
    template<typename Individual>
    class Population
    class Population
    {
        std::vector<Individual> individuals;
        Individual* fittest;

    public:
        Population(size_t size = 0)
            : individuals{ }, fittest{ nullptr }
        {
            if (size)
            {
                individuals.resize(size);
                updateFittest();
            }
        }

        size_t getSize() const
        {
            return individuals.size();
        }

        Individual& getIndividual(size_t i)
        {
            return individuals[i];
        }

        Individual* getFittest() const
        {
            return fittest;
        }

        /**********************************************************************
        \brief
        Update Fittest function to update fitness if there is a higher fitness
        value than previous

        \return
        None
        **********************************************************************/
        void updateFittest()
        {
            if (this->individuals.size())
            {
                this->fittest = &individuals[0];

                for (size_t i = 1; i < this->individuals.size(); ++i)
                {
                    if (individuals[i].getFitness() > fittest->getFitness())
                    {
                        this->fittest = &individuals[i];
                    }
                }
            }
            else
                this->fittest = nullptr;
        }

        friend std::ostream& operator<<(std::ostream& os, Population& rhs)
        {
            os << " = " << rhs.getFittest()->getFitness() << std::endl;
            for (size_t i = 0; i < rhs.getSize(); ++i)
                os << "  " << i << ':' << rhs.getIndividual(i) << std::endl;
            return os;
        }
    };

    // Genetic Algorithm class
    template<typename Individual>
    class GeneticAlgorithm
    {
        Population<Individual>* population;
        int generation;

    public:
        GeneticAlgorithm(): population{ nullptr }, generation{ 0 }{}

        /**********************************************************************
        \brief
        Destructor for class GeneticAlgorithm

        \return
        None
        **********************************************************************/
        ~GeneticAlgorithm()
        {
            if(population)
                delete population;
        }

        /**********************************************************************
        \brief
        Getter function for getFittest()

        \return
        Returns a pointer to Class Individual
        **********************************************************************/
        Individual* getFittest() const
        {
            return this->population->getFittest();
        }

        /**********************************************************************
        \brief
        Implementation of the Roulette Wheel Selection. The probability of an
        individual to be selected is directly propoertional to its fitness.

        \param sizeOfPopulation
        Population size
```

```
        \return
        Returns a pointer to class Population
        ***********************************************************************/
        Population<Individual>* selection(size_t sizeOfPopulation)
        {
            if (!this->population)
                this->setPopulation(new Population<Individual>
                    (sizeOfPopulation));

            Population<Individual>* newGeneration =
                new Population<Individual>(sizeOfPopulation);

            //Play roulette
            int sum_fitness = 0;

            for (size_t i = 0; i < sizeOfPopulation; ++i)
                sum_fitness += this->population->getIndividual(i).getFitness();

            for (size_t i = 0; i < sizeOfPopulation; ++i)
            {
                int random = std::rand() % sum_fitness;

                size_t j = 0;

                while(j < sizeOfPopulation)
                {
                    random -= this->population->getIndividual(j).getFitness();

                    if (random <= 0)
                        break;

                    ++j;
                }

                //Copy genes
                newGeneration->getIndividual(i).copyGenesFrom
                (this->population->getIndividual(j));
            }

            newGeneration->updateFittest();
            return newGeneration;
        }

        /***********************************************************************
        \brief
        Crossover parents genes function

        \param newGeneration
        A pointer to class Population

        \param crossoverMethod
        An enumeration

        \return
        None
        ***********************************************************************/
        void crossover(Population<Individual>* newGeneration,
            CrossoverMethod crossoverMethod)
        {
            int crossOverPoint = 0;

            if (crossoverMethod == CrossoverMethod::Middle)
                crossOverPoint = Individual::chromosome_type::size / 2;
            else if (crossoverMethod == CrossoverMethod::Random)
                crossOverPoint =
                std::rand() % Individual::chromosome_type::size;

            for (size_t j = 0; j < newGeneration->getSize() - 1; j+=2)
            {
                //Swap values among pairs
                for (int i = 0; i < crossOverPoint; ++i)
                {
                    auto t = newGeneration->getIndividual(j + 1).getGene(i);

                    newGeneration->getIndividual(j + 1).setGene(i,
                        newGeneration->getIndividual(j).getGene(i));

                    newGeneration->getIndividual(j).setGene(i, t);
                }
            }
        }

        /***********************************************************************
        \brief
        Mutation of genes under a random probability

        \param newGeneration
        A pointer to class Population

        \param mutationProbability
        Percentatage of mutation

        \return
        None
        ***********************************************************************/
        void mutation(Population<Individual>* newGeneration,
            int mutationProbability)
        {
            int sizeOfPopulation = newGeneration ?
                                    newGeneration->getSize() : 0;

            //Select a random mutation point and flip
            //gene at the mutation point
            for (int j = 0; j < sizeOfPopulation; ++j)
            {
                if (std::rand() % 100 < mutationProbability)
                    newGeneration->getIndividual(j).getChromosome().
                    randomMutation();
                    randomMutation();
            }
        }

        /***********************************************************************
        \brief
        Replace existing population with a new generation

        \param newGeneration
        A pointer to class Population

        \return
        None
        ***********************************************************************/
        void setPopulation(Population<Individual>* newGeneration)
        {
            if (population)
                delete population;

            this->population = newGeneration;
            this->population->updateFittest();
        }

        /***********************************************************************
        \brief
        Start the search

        \param sizeOfPopulation
        Population size

        \param mutationProbability
        Percentage of mutation

        \param crossoverMethod
        An Enumeration

        \param os
        Output stream

        \return
        None
        ***********************************************************************/
        void run(size_t sizeOfPopulation = 100, int mutationProbability = 70,
                    CrossoverMethod crossoverMethod = CrossoverMethod::Middle,
                    std::ostringstream* os = nullptr)
        {
            this->generation = 0;

            this->setPopulation(new Population<Individual>(sizeOfPopulation));

            //While loop unti the solution is found
            while (this->next(mutationProbability, crossoverMethod, os)) {}
        }

        /***********************************************************************
        \brief
        Continue the search

        \param mutationProbability
        Percentage of mutation

        \param crossoverMethod
        An Enumeration

        \param os
        Output stream

        \return
        Return true if solution found, else return false if not found.
        ***********************************************************************/
        bool next(int mutationProbability, CrossoverMethod crossoverMethod,
                    std::ostringstream* os)
        {
            if(!this->population)
                this->setPopulation(new Population<Individual>
                    (this->population->getSize()));

            Individual* fittest = this->population->getFittest();

            if (os)
                *os << *fittest;

            //Stop the search when either max fitness of solution or maxi limit
            //for generation achieved
            if (this->population->getFittest()->getFitness() == 100 ||
                this->generation > 10000)
                return false;

            Population<Individual>* newGeneration =
                this->selection(this->population->getSize());

            //Recombination: creates new individuals by taking the chromosomes
            //from the fittest members of the population and modifying these
            //chromosomes using crossover and/or mutation.
            this->crossover(newGeneration, crossoverMethod);
            this->mutation(newGeneration, mutationProbability);

            //Set population with new generation
            this->setPopulation(newGeneration);
            this->generation++;

            return true;
        }
    };

} // end namespace

#endif
```

◄ Showcase: Genetic Algorithm

Jump to... ⬍

Example questions for Genetic Algorithm with answers for quiz 4 ►

VPL

You are logged in as Wei Zhe GOH (Log out)
cs380su21-meta.sg
Data retention summary
Get the mobile app