Number Partition Problem Genetic Algorithm

Generated by Doxygen 1.8.12

Contents

Chapter 1

NumberPartitionGA

This is a program to run two algorithms to solve the well-known Number Partition Problem (NPP) The NPP is taking a set of numbers and partition that set. Each element in the original set may be placed in either of the two sub partitions. The goal is to find a solution where the sum of the numbers in each partition is equal. For this particular algorithm, a fitness is represented by an integer. A perfect solution will have a value of 0. As this number increases, the solution is further from a perfect solution. For a particular set, there might not be a perfect solution. A "chromosome" is the term I will use for a particular solution. A chromosome is represented as a binary string of 1's and 0's. A 1 represents a value in one of the two partitions and a 0 represents a value in the other partition. Each position of the bits corresponds to the position of the SORTED ascending original list.

These two algorithms are as follows: A greedy algorithm called the longest processing time heuristic. This algorithm first takes the largest element in the original set and places it in the "left" partition. The sum for that partition is then calculated (one value at this point). The element is then removed from the original set. The repetitive process at this point is to take the highest value element in the original set and place it in the partition with the smaller sum. This is repeated until there are no longer any elements left in the original set.

The genetic algorithm (GA) is an algorithm that takes elements from biological evolution and mimics them to alter chromosomes that have a better fitness than their respective parents. The general steps I take for this algorithm are:

- Creating a finite set of randomly generated chromosomes (initial population)
- Using 2-tournament selection to choose two parents from the population
- Use 2-point crossover to splice the chromosomes from the two parents to create an offspring chromosome
- Simulate genetic mutation by making the probability of each bit in an offspring chromosome be the opposite of what it was. This probability is much higher in this algorithm than actaul biology. This is to help speed up the process. 10% probabilty is a good area to work with.

After the finite number of generations are completed, the algorithm might have found multiple solutions. All of the best solutions are then shown with their respective fitness (abs(sumLeft - sumRight)), binary chromosome, and numeric partitions.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Class Documentation

4.1 GA::finalSolutionStruct Struct Reference

`#include <GA.h>`

Public Attributes

- int fitness
- std::vector< std::string > binarySolution
- std::vector< int > leftPartition
- std::vector< int > rightPartition

4.1.1 Member Data Documentation

4.1.1.1 binarySolution

`std::vector<std::string> GA::finalSolutionStruct::binarySolution`

4.1.1.2 fitness

`int GA::finalSolutionStruct::fitness`

4.1.1.3 leftPartition

`std::vector<int> GA::finalSolutionStruct::leftPartition`

4.1.1.4 rightPartition

`std::vector<int> GA::finalSolutionStruct::rightPartition`

The documentation for this struct was generated from the following file:

- F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/GA.h

4.2 GA Class Reference

`#include <GA.h>`

Collaboration diagram for GA:

Classes

- struct finalSolutionStruct

Public Member Functions

- GA ()

  Default constructor for genetic algorithm. This should not be used by default.
- GA (std::vector< int > startingList, int INIT_POP_SIZE)

  Constructor for genetic algorithm.
- ~GA ()

  Default deconstructor for GA class.
- void run (int itterations)

  run executes the genetic algorithm.
- std::vector< std::string > tournament (std::vector< std::vector< std::string >> &L, int &popSize)

  tournament picks two randomly selected parents from the population. The fitness of each chromosome is compared. The one with the lowest fitness (better fitness) is the chosen and placed back into the population. The loser chromosome is then discarded from the popultion.
- void displayPopulation (const std::vector< std::vector< std::string >> vector_const)

  Displays to the console each chromosome and their binary representation. This can be called at any time to view this information.
- int getFitness (std::vector< std::string > chromosome)
- std::vector< int > getSortedList ()
- void setBestSolution (std::vector< std::vector< std::string >> &L)

  Compares all of the chromosomes in the current population and picks the first best solution from the population. This is then stored into a data structure called finalSolution.

Public Attributes

- struct GA::finalSolutionStruct finalSolution

4.2.1 Constructor & Destructor Documentation

4.2.1.1 GA() [1/2]

```
GA::GA ( )
```

Default constructor for genetic algorithm. This should not be used by default.

4.2.1.2 GA() [2/2]

```
GA::GA (
            std::vector< int > startingList,
            int popsize )
```

Constructor for genetic algorithm.

Parameters

startingList	A vector which contains the origonal number set
popsize	<INT> How many chromosomes are generated and kept in the population

4.2.1.3 ~GA()

```
GA::~GA ( )
```

Default deconstructor for GA class.

4.2.2 Member Function Documentation

4.2.2.1 displayPopulation()

```
void GA::displayPopulation (
            const std::vector< std::vector< std::string >> vector_const )
```

Displays to the console each chromosome and their binary representation. This can be called at any time to view this information.

Parameters

vector_const	The vector containing the population of chromosomes. This is constant because we are only viewing data.

Returns

void

4.2.2.2 getFitness()

```
int GA::getFitness (
            std::vector< std::string > chromosome )
```

4.2.2.3 getSortedList()

```
std::vector< int > GA::getSortedList ( )
```

4.2.2.4 run()

```
void GA::run (
            int itterations )
```

run executes the genetic algorithm.

Parameters

itterations	<INT> How many time the algorithm will loop (number of generations)

Returns

void

Here is the caller graph for this function:

4.2.2.5 setBestSolution()

```
void GA::setBestSolution (
            std::vector< std::vector< std::string >> & L )
```

Compares all of the chromosomes in the current population and picks the first best solution from the population. This is then stored into a data structure called finalSolution.

Parameters

L	a vector containing the current population of chromosomes

Returns

void

4.2.2.6 tournament()

```
std::vector< std::string > GA::tournament (
            std::vector< std::vector< std::string >> & L,
            int & popSize )
```

tournament picks two randomly selected parents from the population. The fitness of each chromosome is compared. The one with the lowest fitness (better fitness) is the chosen and placed back into the population. The loser chromosome is then discarded from the popultion.

Parameters

L	a vector containing the current chromosome population
popSize	current population size vector<string> Returns the winner chromosome from the tournament selection

4.2.3 Member Data Documentation

4.2.3.1 finalSolution

```
struct GA::finalSolutionStruct GA::finalSolution
```

The documentation for this class was generated from the following files:

- F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/GA.h
- F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/GA.cpp

4.3 Greedy Class Reference

`#include <Greedy.h>`

Public Member Functions

- Greedy ()
- ~Greedy ()

  Default deconstructor for Greedy.
- void run (std::vector< int > &L, bool displayTwoLists)

  Starts the main algorithm for the greedy algorithm.
- int getFitness ()

4.3.1 Constructor & Destructor Documentation

4.3.1.1 Greedy()

```
Greedy::Greedy ( )
```

Default constructor for Greedy.

4.3.1.2 ~Greedy()

```
Greedy::~Greedy ( )
```

Default deconstructor for Greedy.

4.3.2 Member Function Documentation

4.3.2.1 getFitness()

```
int Greedy::getFitness ( )  [inline]
```

Here is the caller graph for this function:

4.3.2.2 run()

```
void Greedy::run (
            std::vector< int > & L,
            bool displayTwoLists )
```

Starts the main algorithm for the greedy algorithm.

Parameters

L	is a vector containing the origonal number set
displayTwoLists	<bool> Displays extra information about the final results

Returns

void

Here is the caller graph for this function:

The documentation for this class was generated from the following files:

- F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/Greedy.h
- F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/Greedy.cpp

Chapter 5

File Documentation

5.1 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/GA.cpp File Reference

```
#include "stdafx.h"
#include "GA.h"
#include <list>
#include <time.h>
#include <iostream>
#include <string>
#include <vector>
#include <functional>
#include <algorithm>
#include <random>
```

Include dependency graph for GA.cpp:

5.2 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/GA.h File Reference

```
#include <vector>
```

Include dependency graph for GA.h: This graph shows which files directly or indirectly include this file:

Classes

- class GA
- struct GA::finalSolutionStruct

5.3 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/Greedy.cpp File Reference

```
#include "stdafx.h"
#include "Greedy.h"
#include <list>
#include <iostream>
#include <string>
#include <functional>
#include <vector>
#include <algorithm>
```

Include dependency graph for Greedy.cpp:

5.4 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/Greedy.h File Reference

```
#include <list>
#include <string>
#include <vector>
```

Include dependency graph for Greedy.h: This graph shows which files directly or indirectly include this file:

Classes

- class Greedy

5.5 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/NumberPartitionGA.cpp File Reference

Use a greedy algorithm and genetic algorithm to solve the number partition problem.

```
#include "stdafx.h"
#include <iostream>
#include <list>
#include <time.h>
#include <string>
#include <algorithm>
#include "Greedy.h"
#include "GA.h"
#include <vector>
#include <conio.h>
```

Include dependency graph for NumberPartitionGA.cpp:

Functions

- void check ()

  Keeps the console from closing after main has finished.
- void helpMenu ()

  Displays the help menu to the console.
- int main (int argc, char ∗argv[ ])

5.5.1 Detailed Description

Use a greedy algorithm and genetic algorithm to solve the number partition problem.

Author

Steven Scholz

Date

9/21/17

Version

1.0

5.5.2 DESCRIPTION

This is a program to run two algorithms to solve the well-known Number Partition Problem (NPP) The NPP is taking a set of numbers and partition that set. Each element in the original set may be placed in either of the two sub partitions. The goal is to find a solution where the sum of the numbers in each partition is equal. For this particular algorithm, a fitness is represented by an integer. A perfect solution will have a value of 0. As this number increases, the solution is further from a perfect solution. For a particular set, there might not be a perfect solution. A "chromosome" is the term I will use for a particular solution. A chromosome is represented as a binary string of 1's and 0's. A 1 represents a value in one of the two partitions and a 0 represents a value in the other partition. Each position of the bits corresponds to the position of the SORTED ascending original list.

These two algorithms are as follows: A greedy algorithm called the longest processing time heuristic. This algorithm first takes the largest element in the original set and places it in the "left" partition. The sum for that partition is then calculated (one value at this point). The element is then removed from the original set. The repetitive process at this point is to take the highest value element in the original set and place it in the partition with the smaller sum. This is repeated until there are no longer any elements left in the original set.

The genetic algorithm (GA) is an algorithm that takes elements from biological evolution and mimics them to alter chromosomes that have a better fitness than their respective parents. The general steps I take for this algorithm are:

- Creating a finite set of randomly generated chromosomes (initial population)
- Using 2-tournament selection to choose two parents from the population
- Use 2-point crossover to splice the chromosomes from the two parents to create an offspring chromosome
- Simulate genetic mutation by making the probability of each bit in an offspring chromosome be the opposite of what it was. This probability is much higher in this algorithm than actaul biology. This is to help speed up the process. 10% probabilty is a good area to work with.

After the finite number of generations are completed, the algorithm might have found multiple solutions. All of the best solutions are then shown with their respective fitness (abs(sumLeft - sumRight)), binary chromosome, and numeric partitions.

You can use the argument -h to call the help menu for the different command line arguments

5.5.3 Function Documentation

5.5.3.1 check()

```
void check ( )
```

Keeps the console from closing after main has finished.

Returns

void

Here is the caller graph for this function:

5.5.3.2 helpMenu()

```
void helpMenu ( )
```

Displays the help menu to the console.

Returns

void

Here is the caller graph for this function:

5.5.3.3 main()

```
int main (
            int argc,
            char * argv[] )
```

Here is the call graph for this function:

5.6 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/stdafx.cpp File Reference

```
#include "stdafx.h"
```
Include dependency graph for stdafx.cpp:

5.7 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/stdafx.h File Reference

```
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
```
Include dependency graph for stdafx.h: This graph shows which files directly or indirectly include this file:

5.8 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/NumberPartitionGA/targetver.h File Reference

```
#include <SDKDDKVer.h>
```

Include dependency graph for targetver.h: This graph shows which files directly or indirectly include this file:

5.9 F:/Google Drive/CSCI443/Project 2/NumberPartitionGA/README.md File Reference

Index