Genetic Algorithm

Subset Sum Problem

Team Member:
Han Luo 001813558
Xianlong He 001818123

April 15th, 2018

Introduction

1.1 Problem Description

This project is trying to solve an NP-Complete problem named Subset Sum Problem and provide this problem with a “good enough” solution.

Subset sum problem describes a situation that given a set of integers, does this set have a non-empty subset which can sum up to zero. This problem is an NP-complete problem which means it is easy to confirm wether the proposed answer is valid. However, it is basically impossible to determine if there is any solution from the very beginning.

Given a set {1, 7, 4, -5, -3}, the subset sum problem regarding to this set has a answer {1, 4, -5}. This might be an easy situation, if the size of a set is a lot bigger, the situation would not be this clear, and be prohibitively difficult to determine whether there is a solution to this set.

Data structures and Methods

2.1 Data Structures

In this Program the main data structure writers are using is called “Candidate”. In the following form is the introduction of all the properties of Candidate.

Candidate( )	Basic data structure has two inputs: genotype and original set. Implements Comparable for easily compare purpose
Geno	Type int[ ] property, used to store gene consists by 1s and 0s
Fit	Type int property that defines how candidate fits with the environment

2.2 Function Description

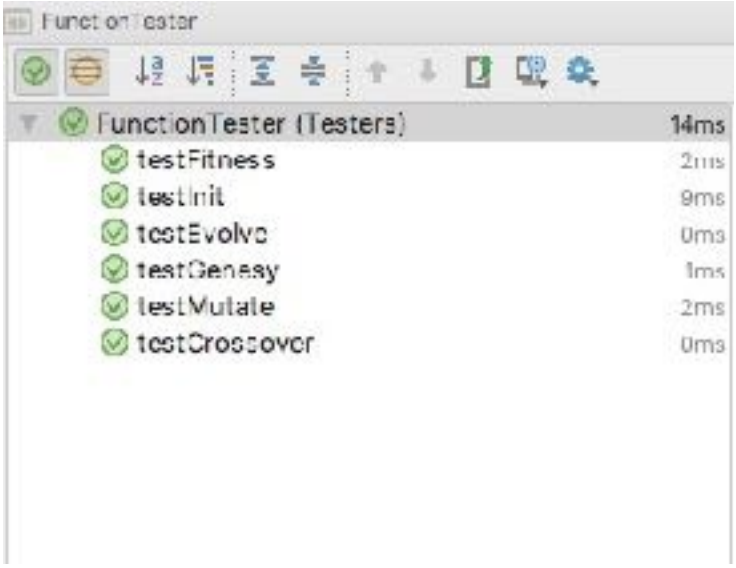
Design for this program is to use a series of 1s and 0s to represent as Genotype of an individual and it also has the same length of the original set, the original set of integers represents the environment. The phenotype of an individual is when genotype and the original set multiply, and produce multiple combinations of integer sets. This was to simulate the subsets of the original set.

After the presentations of Genotype and Phenotype are settled comes the program flow. Implementation is straightforward, the original set of integers are generated separately, determine the group of candidates that are fit, make the fit ones crossover and mutate by chance to breed next generation. Keep track of the best one in the entire generation and record them.

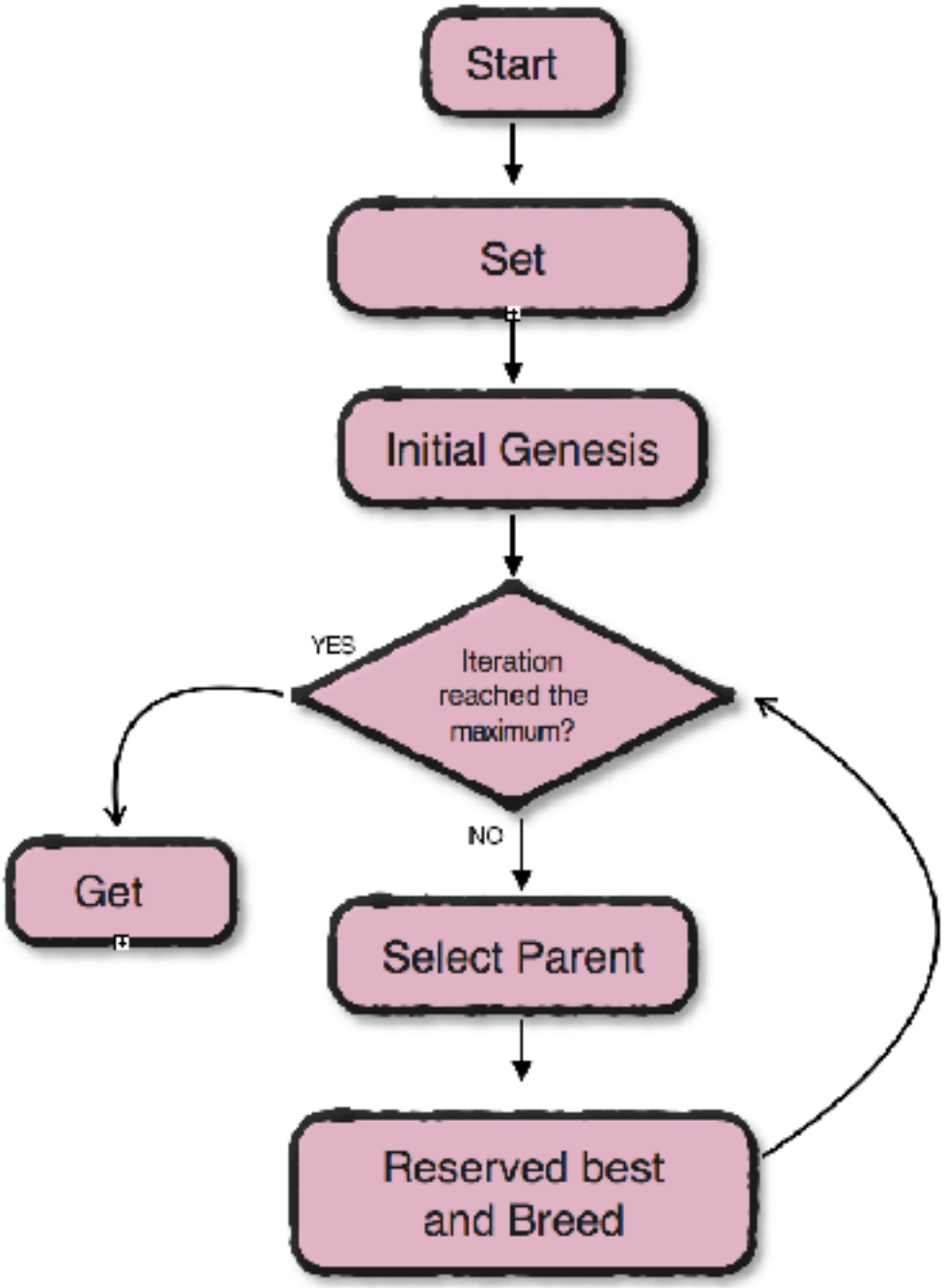
Function	Description
fitness( )	Calculates the fit property of a candidate object, it is in class Candidate and called every time a calculate object is created. It multiplies the “geno” integer string with “raw” the original set and produces the sum of every exist integer which is the fit of this candidate
init( )	Used to generate genotype, using Math.random to generate a series of 1s and 0s
genesy( )	Creates the original set called “raw”, the method used here is Math.random it generates a certain amount of integers within the range of -100 to 1000. The first generation of genotype using init( ). Take “raw” and “geno” as an input parameter and create new candidate. Keep repeating the process and add it to an ArrayList of Candidates. This ArrayList is the first generation of candidates which is the output of this function.
mutate( )	Take an integer array as an input and change the content for the first part of the array. In this case, it changes 1 to 0, and 0 to 1
crossover( )	Randomly take two candidates of the parent generation. Each position of the decedent’s gene integer array has a possibility of inheriting from either father gene or mother gene. The percentage is implemented using Math.random. In this way, we will set a new Array of Integer to generate a child candidate. On the other hand, when a candidate has generated it has a possibility of mutating. We use parameter “pro” to adjust the possibility.
evolve( )	Decide which candidates survive the environment, input parameters are a generation of candidates and a number from 0 to 100 which indicates the percentage that user intends to keep of this generation. Output is the remaining candidates.
breed( )	Provides the new generation of candidates. Produce the same amount of candidates as the previous generation. Apply function crossover to breed new candidate one by one and return an ArrayList of candidates

2.3 Unit Test

Unit test class consists 6 unit tests covers all methods. And all of them passed the unit tests.



2.4 The WorkFlow Of Program



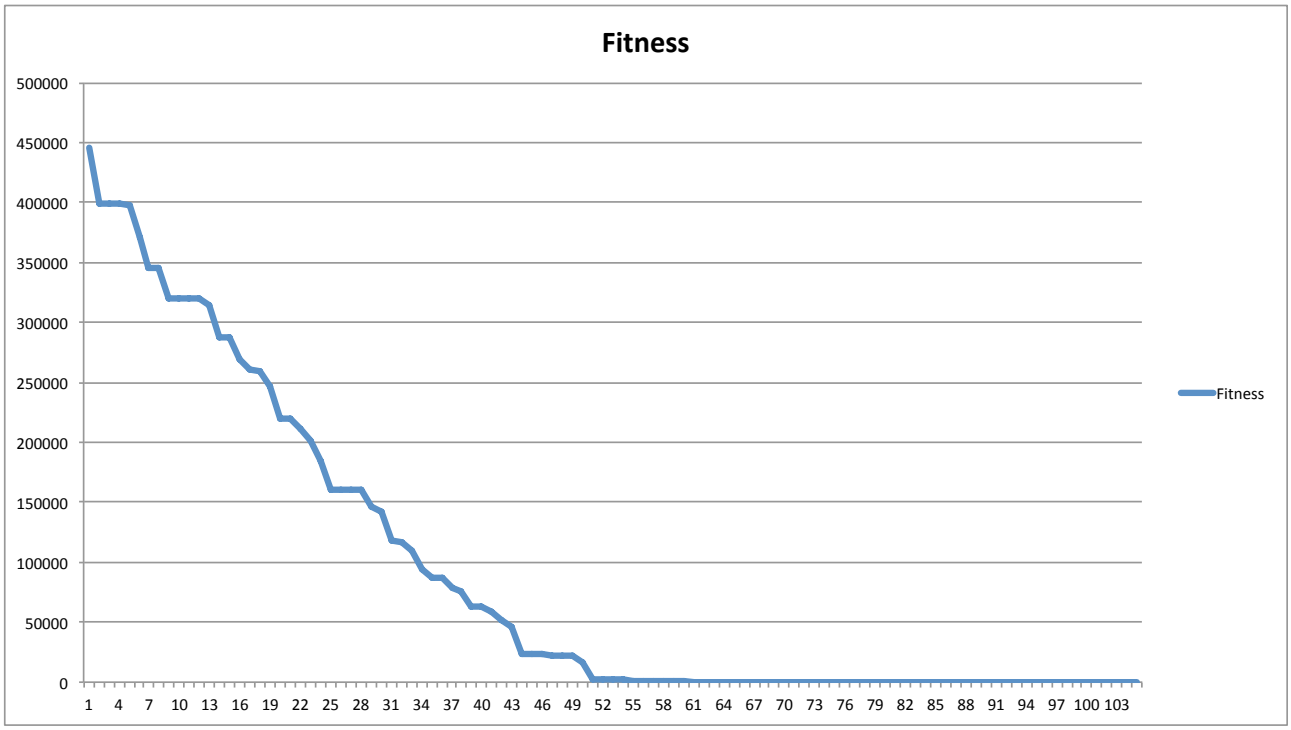
Test and Result

Generation 1:445322	Generation 36:86828	Generation 71:1
Generation 2:399279	Generation 37:78725	Generation 72:1
Generation 3:399279	Generation 38:76427	Generation 73:1
Generation 4:399279	Generation 39:63158	Generation 74:1
Generation 5:397822	Generation 40:63158	Generation 75:1
Generation 6:371560	Generation 41:58606	Generation 76:1
Generation 7:346096	Generation 42:52402	Generation 77:1
Generation 8:346096	Generation 43:45573	Generation 78:1
Generation 9:320603	Generation 44:24094	Generation 79:1
Generation 10:320603	Generation 45:24094	Generation 80:1
Generation 11:320603	Generation 46:24094	Generation 81:1
Generation 12:320603	Generation 47:22244	Generation 82:1
Generation 13:314744	Generation 48:22244	Generation 83:1
Generation 14:287249	Generation 49:22244	Generation 84:1
Generation 15:287249	Generation 50:16999	Generation 85:1
Generation 16:269893	Generation 51:2746	Generation 86:1
Generation 17:260173	Generation 52:2746	Generation 87:1
Generation 18:258685	Generation 53:2746	Generation 88:1
Generation 19:246445	Generation 54:2746	Generation 89:1
Generation 20:219498	Generation 55:637	Generation 90:1
Generation 21:219498	Generation 56:637	Generation 91:1
Generation 22:211887	Generation 57:249	Generation 92:1
Generation 23:201769	Generation 58:249	Generation 93:1
Generation 24:184965	Generation 59:249	Generation 94:1
Generation 25:160833	Generation 60:249	Generation 95:1
Generation 26:160833	Generation 61:49	Generation 96:1
Generation 27:160833	Generation 62:7	Generation 97:1
Generation 28:160833	Generation 63:7	Generation 98:1
Generation 29:146828	Generation 64:7	Generation 99:1
Generation 30:142014	Generation 65:7	Generation 100:1
Generation 31:117445	Generation 66:7	Generation 101:1
Generation 32:116882	Generation 67:7	Generation 102:1
Generation 33:109562	Generation 68:7	Generation 103:1
Generation 34:94437	Generation 69:1	Generation 104:1
Generation 35:86828	Generation 70:1	Generation 105:0

3.1 Analysis

From *Fitness chart* below, The Fit value of very first generations plunges to a relatively low level. The turning point first appears around the 50th generation. Then value becomes stable for a long period of time until the 105th generation. At the 105th generation, Our 'good enough' appeared. It represents the best case only in a specific interval of solutions. This interval would be determined by the first generation of Parent ArrayList.

This result is perfectly consistent with our expectation. According to our Algorithm structure, The plunge is derived from Crossover which effectively keeps the desired gene. However, Crossover also lead the population towards the stable. In order to make more progress, the Mutation takes the most weight than Crossover.



Fitness Chart

Output of best generation:

{-2 -255 7465 -371 7704 1419 -149 6840 5542 4585 6741 -606 -409 2354 -483 -169 1759 -93 -730 -228 -445 -796 -155 -524 -236 -553 -598 -897 -380 -40 -140 -967 1423 -631 -470 -460 -854 -150 -919 -697 -635 887 -923 -325 -226 -521 -765 -893 -697 81 -3 -58 -437 2763 -142 -891 -867 -208 -996 -145 -66 -934 -97 -541 54 -543 -10 -983 -13 -359 -927 -997 -649 0 -52 -714 -339 -797 -881 -981 -419 -303 -699 -827 -643 1150 164 3276 -162 -510 -907 -685 -858 -440 -93 -927 982 -267 -562 2864 -861 -380 -82 87 -234 -736 -678 -691 2273 -834 -562 -975 -126 -92 15 -485 -813 -949 -828 -484 -635 -734 -367 -31 1536 -924 -706 -535 -636 -798 -522 -146 -588 -459 771 -732 -24 -663 -360 -812 2141 -283 -599 1192 -18 -246 -458 -400 -701 -411 -89 -898 -985 -75 -225 -808 -713 -674 1981 -217 1518 -151 -45 -827 -938 }