Project 6
"Op" Philip Flarsheim
Computer Engineering & Computer Science
Speed School of Engineering
University of Louisville, USA
plflar02@louisville.edu

1. Introduction

This project was an investigation in the combination of "Genetic Algorithm" and "Wisdom of Crowds" strategies to solve the Partition Sum NP complete problem. The Partition Sum problem consists of a set of n integers with values ranging from $1 \rightarrow k$. These object is to determine if there are two subsets whose sums are equal. This problem was selected for investigation since its solution mapped well to the idea of a genome in Genetic Algorithms.

2. Approach

Genetic Algorithms take their inspiration from the biological theory of evolution. In this instance the *'genome'* was defined as two lists of numbers containing numbers from the original list, with each number being used once across both lists. Each *'genome'*s fitness was calculated as the difference between the sums of the two lists. The initial population was created by shuffling the array of starting integers and then splitting it at a random point into two lists. The starting population was split into several groups and were allowed to evolve in isolation for several generation; followed by a step in which randomly selected members of each population were shared with other populations. Each generation consisted of several discrete steps. First the sorted population was separated into groups each consisting of one ninth of the population and the first, third and fifth groups were taken to be the base population for the next generation. These members were then crossed at random with any of the members of the previous generation, each union resulting in two children each with a different dominant parent genome, those children were added to the new population. Second a percent chance was rolled for each member of the new population to determine if several of its genes where randomly switched. The percent chance of mutation was determined as 5% plus 1% for every generation in which the same member of the population was the fittest. This averaged mutation chance at about 10%.

The Wisdom of Crowds search strategy makes use of swam intelligence by aggregating a *'crowd'* and building a best solution off of each *'expert's'* proposed solution. The set of experts was created by taking the top 10% of all final solutions in the genetic algorithm. A weight system was created to consolidate the positions of each number in either the first or second list. A greedy selection method was used to build the final aggregate solution. The number of times each gene appeared in a list determined which list it was placed in for the final solution.

Algorithm 1 Genetic Algorithm - Main

$populations = randomPopulations(numPops, popSize)$
for $i = 0 \rightarrow numIterations$ **do**
  for all $population \in populations$ **do**
    $population = EvolvePopulation(population, numGenerations)$
  end for
  $populations = MergePopulations(populations, random.int(1, popSize/numPops), random.int(1, numPops-1))$
end for
$Experts = populations.getTop10Percent()$
$Final = WisdomOfCrowds(Experts)$

Algorithm 2 Genetic Algorithm - Evolve Population

```
function EvolvePopulation(population, numGenerations)
    for i = 0 → numGenerations do
        newPop = population.select9nths(1, 3, 5)
        for all parent ∈ newPop do
            newPop.append(Crossover(parent, population.randomChoice()))
            newPop.append(Crossover(random.choice(population), parent))
        end for
        population = newPop
        population.sortByFitness()
        mutateChance = leaderCount * .01 + .05
        for all genome ∈ population do
            if Random() <= mutateChance then
                Mutate(genome)
            end if
        end for
        if population[0] == leader then
            leaderCount + +
        else
            leader = population[0]
            leaderCount = 0
        end if
    end for
    return population
end function
```

Algorithm 3 Genetic Algorithm - Crossover

```
function CROSSOVER(parent1, parent2)
    child = clone(parent1)
    if parent2.list1.length > 0 then
        for i = 0 → random.int(0, parent2.list1.length) do
            value = random.choice(parent2.list1)
            if (child.list2.count(value) > 0) then
                child.list2.remove(value)
                child.list1.append(value)
            end if
        end for
    end if
    if parent2.list2.length > 0 then
        for i = 0 → random.int(0, parent2.list2.length) do
            value = random.choice(parent2.list2)
            if (child.list1.count(value) > 0) then
                child.list1.remove(value)
                child.list2.append(value)
            end if
        end for
    end if
    return child
end function
```

Algorithm 4 Genetic Algorithm - Mutate

```
function MUTATE(genome, numSwaps)
    for i = 0 → numSwaps do
        if random() < .5 then
            if (genome.list1.length > 0 then
                value = random.choice(genome.list1)
                genome.list1.remove(value)
                genome.list2.append(value)
            end if
        else
            if (genome.list2.length > 0 then
                value = random.choice(genome.list2)
                genome.list2.remove(value)
                genome.list1.append(value)
            end if
        end if
    end for
    return genome
end function
```

Algorithm 5 Genetic Algorithm - Merge Populations

```
function MERGEPOPULATIONS(populations, numNomads, direction)
    for i = 0 → populations.length() do
        for j = 0 → numNomads do
            nomad = random.choice(populations[i])
            populations[i].remove(nomad)
            nomads[i].append(nomad)
        end for
    end for
    for i = 0 → nomadss.length() do
        j = i + direction
        if j >= populations.length() then
            j− = populations.length()
        end if
        populations[j].extend(nomad)
    end for
    return populations
end function
```

Algorithm 6 Wisdom of Crowds

```
function WisdomOfCrowds(baseGenome, experts)
    baseGenome.sort()
    for all expert ∈ experts do
        for i = 0 → baseGenome.length() do
            if expert.list1.count(baseGenome[i]) > 0 then
                weight1[i]+ = 1
                expert.list1.remove(baseGenome[i])
            end if
            if expert.list2.count(baseGenome[i]) > 0 then
                weight2[i]+ = 1
                expert.list2.remove(baseGenome[i])
            end if
        end for
    end for
    for i = 0inbaseGenome.length() do
        if weight1[i] > weight2[i] then
            final.list1.append(baseGenome[i])
        else if weight1[i] < weight2[i] then
            final.list2.append(baseGenome[i])
        else
            if random() < .5 then
                final.list1.append(baseGenome[i])
            else
                final.list2.append(baseGenome[i])
            end if
        end if
    end for
    return final
end function
```

3. Results

The algorithm performed significantly better than anticipated, both in speed of executed and quality of results. Several trial runs were performed in order to fine tune the algorithm's initial conditions.

3.1 Data

The initial number set used in the trial was randomly generated by Python's Random library. Each number was an integer between the values of one and one million. The numbers used for the presented trial are presented below.

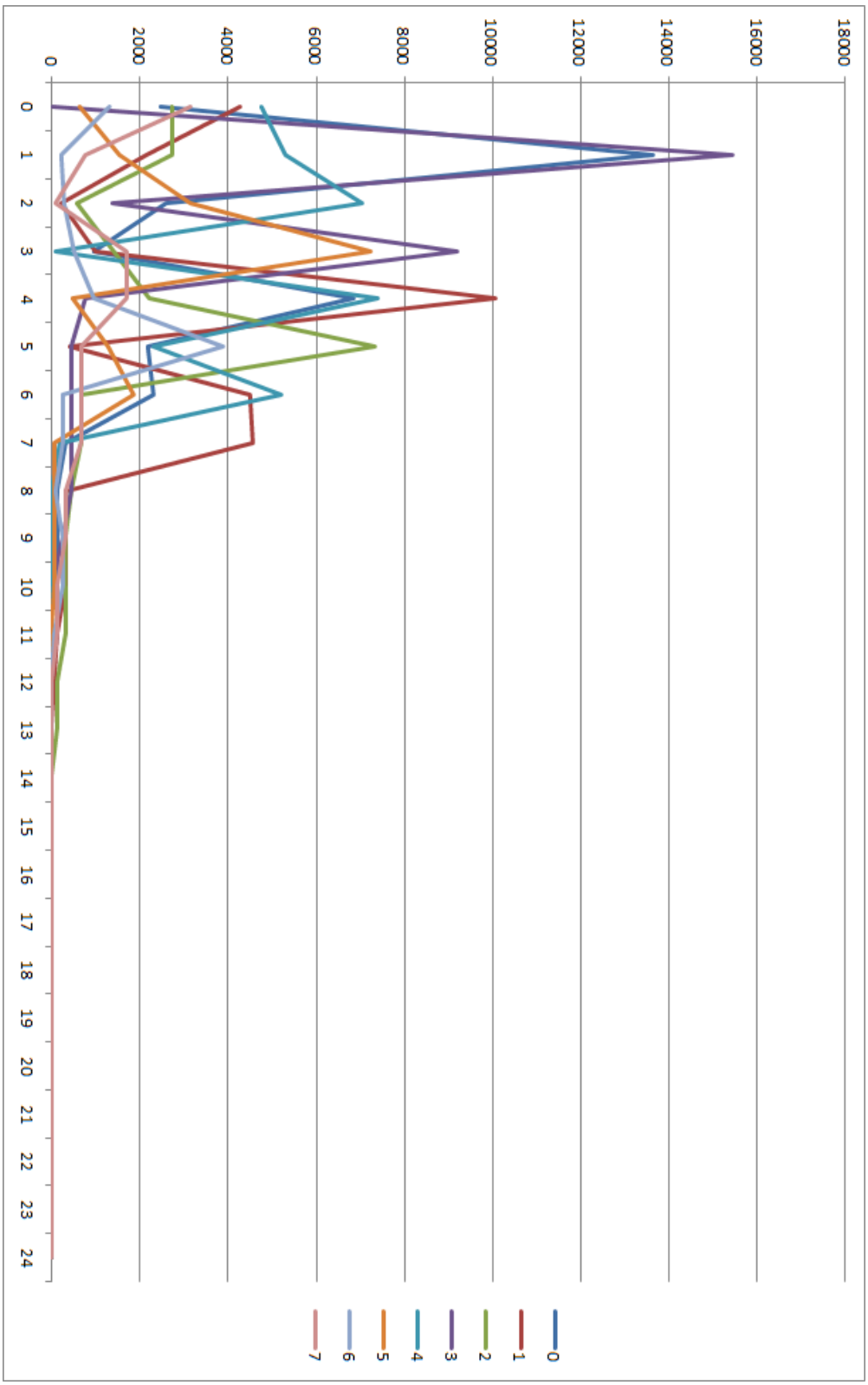
530269, 486126, 189697, 908315, 598817, 513323, 850193, 274382, 156932, 592172, 720857, 420176, 262184, 602360, 214173, 655394, 230661, 112054, 680370, 540869, 865957, 749914, 575825, 196120, 740144, 709610, 611960, 259650, 45305, 800570, 280931, 695019, 666370, 616321, 276223, 740622, 365003, 312588, 7108, 288219, 313276, 925266, 389314, 135313, 460071, 515883, 627147, 486405, 594214, 972517, 693666, 334514, 812318, 400648, 306611, 99961, 480851, 645931, 945743, 172982, 752005, 365494, 757666, 301042, 704655, 822154, 536762, 77360, 838537, 289654, 475082, 54531, 937713, 100403, 91491, 971592, 335637, 436072, 807055, 370362, 470626, 914511, 527257, 568457, 887690, 114547, 868227, 224249, 724245, 308333, 85901, 122762
901593, 112463, 218622, 319587, 669046, 618715, 520159, 699781,

3.2 Results

Initial conditions	
Genome Size	100
Number of Iterations	25
Generations Per Iteration	100
Number of Populations	8
Population Size	250

Results	
Time(s)	834.3586609363556
Sum Difference	2
List1	7108, 45305, 54531, 77360, 99961, 112054, 112463, 114547, 122762, 135313, 156932, 189697, 230661, 259650, 262184, 289654, 301042, 308333, 370362, 389314, 400648, 420176, 436072, 475082, 486126, 486405, 515883, 536762, 568457, 575825, 598817, 627147, 645931, 669046, 680370, 699781, 709610, 720857, 724245, 740144, 740622, 757666, 800570, 812318, 838537, 850193, 901593, 908315, 914511, 925266, 972517
List2	85901, 91491, 100403, 172982, 196120, 214173, 218622, 224249, 274382, 276223, 280931, 288219, 306611, 312588, 313276, 319587, 334514, 335637, 365003, 365494, 460071, 470626, 480851, 513323, 520159, 527257, 530269, 540869, 592172, 594214, 602360, 611960, 616321, 618715, 655394, 666370, 693666, 695019, 704655, 749914, 752005, 807055, 822154, 865957, 868227, 887690, 937713, 945743, 971592

3.2.1 Genetic Algorithm Results



3.2.2 Weight Results

Gene	List1	List2
7108	486	0
45305	486	0
54531	486	0
77360	486	0
85901	0	486
91491	0	486
99961	486	0
100403	0	486
112054	486	0
112463	486	0
114547	486	0
122762	486	0
135313	486	0
156932	486	0
172982	0	486
189697	486	0
196120	0	486
214173	0	486
218622	0	486
224249	0	486
230661	486	0
259650	486	0
262184	486	0
274382	0	486
276223	0	486
280931	0	486
288219	0	486
289654	486	0
301042	486	0
306611	0	486
308333	486	0
312588	0	486
313276	0	486
319587	0	486

Gene	List1	List2
334514	0	486
335637	0	486
365003	0	486
365494	0	486
370362	486	0
389314	486	0
400648	486	0
420176	486	0
436072	486	0
460071	0	486
470626	0	486
475082	486	0
480851	0	486
486126	486	0
486405	486	0
513323	0	486
515883	486	0
520159	0	486
527257	0	486
530269	0	486
536762	486	0
540869	0	486
568457	486	0
575825	486	0
592172	0	486
594214	0	486
598817	486	0
602360	0	486
611960	0	486
616321	0	486
618715	0	486
627147	486	0
645931	486	0
655394	0	486

Gene	List1	List2
666370	0	486
669046	486	0
680370	486	0
693666	0	486
695019	0	486
699781	486	0
704655	0	486
709610	486	0
720857	486	0
724245	486	0
740144	486	0
740622	486	0
749914	0	486
752005	0	486
757666	486	0
800570	486	0
807055	0	486
812318	486	0
822154	0	486
838537	486	0
850193	486	0
865957	0	486
868227	0	486
887690	0	486
901593	486	0
908315	486	0
914511	486	0
925266	486	0
937713	0	486
945743	0	486
971592	0	486
972517	486	0

4. Discussion

The algorithm performed better than expected in both execution time and in quality of results. Genetic algorithms lend themselves significantly better to the Partition Sum problem since the genome does not need to be balanced. This allowed for for effective crossover and faster execution time. Additionally the implementation of isolated populations allowed the algorithm to try multiple paths concurrently. Unfortunately in many of the trial runs the Genetic algorithm was able to calculate the best solution without the need for aggregation through the Wisdom of Crowds algorithm.

5. References

The matplotlib library was used to render the data set and to provide a UI in which to inspect the results. Documentation can be found at http://matplotlib.sourceforge.net/.