A Permutation Based Encoding in a Hybridized Genetic Algorithm for the Bin Packing Problem

Jacob Hopkins
Department of Computer Science
St. Cloud State University
St. Cloud, MN 56301 USA
jrhopkins@stcloudstate.edu

ABSTRACT
The encoding of chromosomes is among the most important design choices of the construction of genetic algorithms (GA). Some problems almost naturally give rise to permutation-based representation. This paper explores the advantages and challenges of encoding candidate solutions of bin packing in permutations. Bin packing involves an arbitrary list of packages with weights within the range of 1 and half of the bin's capacities, who are then analyzed and processed into bins. This is an NP Hard problem. By ordering the list in different sequences we can create different permutations of the packages. Indexing those sequences putting them into bins as they are processed, only creating a new bin when the current bins capacity is full. Decoder functions that maintained permutation integrity were used. Order 1 crossover and a swap mutation ensures that chromosomes are valid, removing the need for discarding, repairing, or penalizing invalid chromosomes. The genetic algorithm written for the use of this paper is on par with the bin count performance of other fit based algorithms but comes with a penalty of speed performance.

Keywords: Bin packing, permutation encoding, order 1 crossover, mutation, genetic algorithms.

INTRODUCTION
Genetic algorithms are robust, reusable, and powerful algorithms for searching large search spaces, and for finding reasonable solutions in reasonable time constraints. The design and inputs to these algorithms are important choices to the success of the algorithm in searching for feasible solutions. GA designers often have several different encoding schemes available and often have to make the selection for encodings conductive to genetic search.

This paper explores the use of permutations for encodings of candidate solutions. Validity of chromosomes is preserved through the use of permutation safe crossover and mutation. Order 1 crossover includes two parent chromosomes, a substring of parent 1 is moved to the child, and those values who are not represented in the child, but still in parent 2 are added to the end of the substring with wrap around to complete the new chromosome containing information from both parents. Mutation is simply the selection of two random indices within the candidate solution and swapping their position. Selection is done by combining the previous and new generation into a mating pool and resupplying the next generation with the top fitness of the mating pool as well as the top 10 chromosomes with the lowest bin count for elitism. The end condition includes a top fitness score that has not changed for 5 generations. Fitness is determined by the sum of the ratios of bin fill count divided by the bin capacity, and the summation divided by the total length of the bins.[2] One would be perfectly filled bins, zero would be completely empty bins, and any value in between represents the fitness of a candidate solution. This fitness is maximized as generations increase.

The following sections of this paper describe bin packing, a genetic algorithm for bin packing, 4 common algorithms for bin packing: Next Fit, Best Fit, Worst Fit, and First Fit, and a comparison of bin count and time performance on different input parameter values: high bin capacity and high packing count, low of both, and either parity.

BIN PACKING
Bin packing is a problem of confining items of various sizes into a finite number of bins B, each with a capacity of b. The finite set of items each have a size of [1, b/2] and are assigned to the minimum number of bins without allowing the total of sizes for each package within a bit to exceed its capacity b. The capacity of the bins and the number of packages to be put into bins is supplied by the user or test case. The set of items with weights are generated randomly between constraints. Example: Suppose there were three packages with weights: [4,5,3]. Suppose we had a bin capacity of 10.
No matter how these objects are grouped, they must take up two bins:[4,5][3], [3,4][5], or [5,3][4] etc. This is bin packing, and those are some of the solutions. Notice that each solution is a permutation of the weights

VARIOUS FIT ALGORITHMS
There exist many traditional algorithms for bin packing. To get a board comparison of the performance of the GA in terms of solution and time.

First Fit (FF)
O(|L| log(|L|)). This algorithm processes packages in an arbitrary list L. For each of those items it attempts to place the package in the first bin in the list of bins that can accommodate it. On the condition that no currently existing bin can accommodate then a new bin will be created with the package inside it.[1]

```
def firstfit(packages, bin_capacity):

    bins = [[]]

    for p in packages:
        for b in bins:
            fill_with_package = fill(b) + p[1]
            if fill_with_package <= bin_capacity:
                b.append(p)
                break
        else:
            bins.append([p])

    return bins
```

- Note these packages have two values: [ID, SIZE]

Best Fit (BF)
O(|L| log(|L|)). This algorithm processes packages in an arbitrary list L. For each of those items it attempts to place the package in the bin with the biggest load still capable of accommodating the package.[1]

```
def bestfit(packages, bin_capacity):

    bins = []

    for p in packages:
        bestfit_bin = []
        min_delta = float('inf')
        for b in bins:
            fill_with_package = (fill(b) + p[1])
            if  fill_with_package <= bin_capacity:
                space_left = bin_capacity - fill_with_package
                if space_left < min_delta:
                    min_delta = space_left
                    bestfit_bin = b

        if bestfit_bin == []:
            bins.append([p])
        else:
            bestfit_bin.append(p)

    return bins
```

Worst Fit (WF)
O(|L| log(|L|)). This algorithm processes packages in an arbitrary list L. For each of those items it attempts to place the package in the bin with the biggest load still capable of accommodating the package.[1]

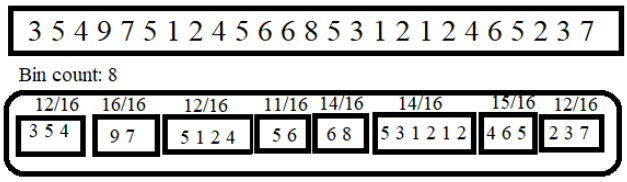
```
def worstfit(packages, bin_capacity):
    bins = []

    for p in packages:
        bestfit_bin = []
        max_delta = float('-inf')
        for b in bins:
            fill_with_package = (fill(b) + p[1])
            if  fill_with_package <= bin_capacity:
                space_left = bin_capacity - fill_with_package
                if space_left > max_delta:
                    max_delta = space_left
                    bestfit_bin = b

        if bestfit_bin == []:
            bins.append([p])
        else:
            bestfit_bin.append(p)

    return bins
```

Next Fit (NF)
O(|L|). This algorithm involves having only one partially filled bin open at a time. Items are considered in order of the list. If an item fits the currently open bin, then it is placed into the bin. If not, the bin is closed and a new bin is opened, and the current item is then placed inside of there. New bins are added when no bin can hold the package.[1]

```
def nextfit(packages, bin_capacity):

    o = 0
    bins = [[]]

    for p in packages:
        placed = False
        while not placed:
            open_bin = bins[o]

            if fill(open_bin) + p[1] <= bin_capacity:
                open_bin.append(p)
                placed = True
            elif o < len(bins) - 1:
                o += 1
            else:
                bins.append([p])
                placed = True

    return bins
```

PERMUTATION CODING
The coding of the present GA is based on the encoding suggested by M Gourgand 2014.[3] Packages are unique items, and therefore can be represented by unique strings, or permutations, giving rise to the order in which packages are processed. Packages are then packaged into bins by taking them in the arbitrary list L and putting them into bins by adjacency until the substring of packages reaches bin capacity, and then a new bin is started. This is a hybridization of the genetic algorithm because we are injecting better chromosomes though the use of a heuristic.

PACKAGE COUNT: 25
BIN CAPACITY: 16



THE GENTIC ALGORITHM

```
GA Structure:
      initialize the population with random chromosomes;
      evaluate all the chromosomes;

      repeat
      {
        for i from 1 to the population size
        {
          decide whether to use crossover or mutation;
          if crossover
            select two parents;
            cross them to produce an offspring;
          else
            select one parent;
            mutate it to produce an offspring;
          evaluate the offspring;
          record the offspring in the next generation;
        }
        offspring replace parents;

      } until enough generations have passed;

      report the solution the best chromosome represents;
```

The permutation coding for bin packing was implemented into a conventional generational genetic algorithm. The population size of the algorithm is 20. The mutation rate of the algorithm is 10% and each chromosome is tested at that rate. If mutated, then the mutagen is added to the new generation. If a chromosome is not mutated, then it undergoes order 1 cross over with a randomly selected chromosome to populate the next generation with their child. There is a 75% replacement rate and the top 25% are kept. The algorithm cuts off when the count of unchanged fitness goes from 0 to 2. Meaning if fitness does not improve for 3 (arbitrarily selected through trial and error for its appeared time and bin performance) generations then the algorithm will end.

Selection

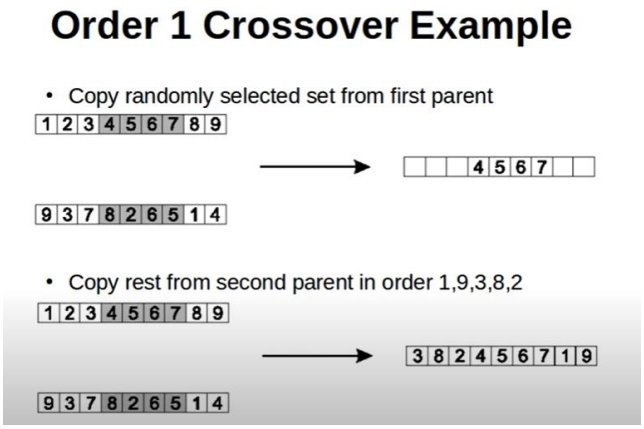
The genetic algorithm used takes the current population and the newly created population and combines them to create a mating pool. This pool is sorted by order of fitness. The top quarter of the last generation with the lowest bin count is passed on as elites. The remainder of the new generation is taken from the top of the mating pool with the highest fitness until the new generation is filled without repeats.

Mutation

Two random mutation points are selected by indices of the chromosome. The two packages at those indices are swapped to produce the mutagen.

Order 1 Crossover

A substring of the first parent is directly copied to the child chromosome. Then those values remaining in the parent who are not represented in the child are added in the order that they are missing starting from the end of the substring and wrapping around. This helps to preserve full presence of sections of parent 1 and preserve mostly the location of packages in the child from parent 2. This also ensures the validity of the child.



Fitness

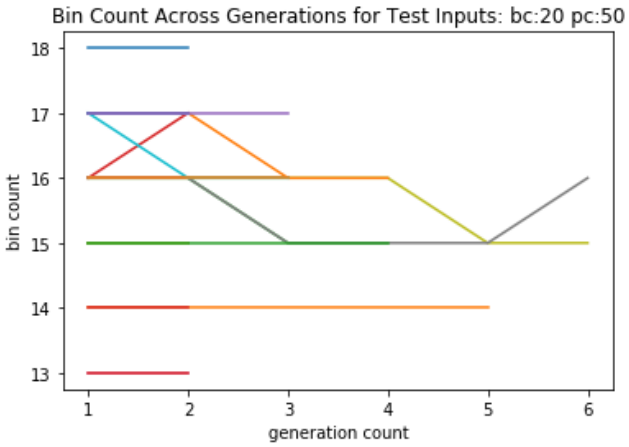
To create a more traversable search space for the genetic search a ratio of the total bin count to the summation of the fill of each bin was used. Given a bin count of n each with capacity c. And a value for k (1 was used in this GA). Take the fill of the bin Fi and divide it by its capacity, sum those ratios and divide it by the bin count. A value of 1 indicates bins are filled completely. A value of 0 would be that there are no bins. Any value in between indicates a fitness, and those closer to 1 are more fit candidate solutions. We are therefore maximizing this value to lower the bin count.

$$F = \frac{\sum_{i=1}^{n} \left(\frac{f_i}{c}\right)^k}{n}$$

COMPARISONS

With two inputs to the problem it makes sense to conduct 4 tests to compare how the parameters affect the algorithms. Therefore, an experiment with high bin capacity and high number of packages, of low for both, and two more of the parity of either parameter was performed. Each test had twenty-five trials, and then the bin count and run time was recorded and averaged. For paper space constraints only 10 are shown.

Low Bin Capacity & Low Number of Packages
With a bin capacity of 20 units, and with 50 packages. Below is the graph of the 25 runs of the genetic algorithm. Each line represents a trails best fitness over time. The graph of fitness is a smoother shallow slope along the top of the graph. Permutations often lead to strong starting solutions.



Average Time(GA): 0.018084096908569335
Average Bin Count(GA): 15.76

Average Time(FF): 0.00016002655029296876
Average Bin Count(FF): 14.84

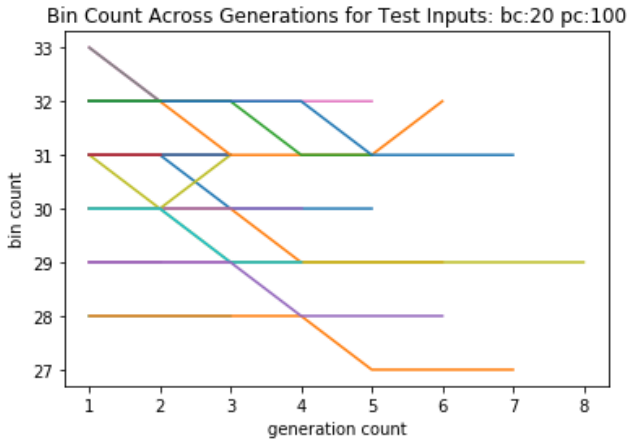
Average Time(BF): 0.00012000083923339844
Average Bin Count(BF): 15.56

Average Time(WF): 4.002571105957031e-05
Average Bin Count(WF): 15.84

Average Time(NF): 4.002571105957031e-05
Average Bin Count(NF): 15.84

	GA	FF	BF	WF	NF
0	15	14	14	14	14
1	14	13	13	14	14
2	15	15	14	15	16
3	17	15	15	16	16
4	16	15	15	16	16
5	15	14	14	15	15
6	13	12	12	12	13
7	18	16	16	17	18
8	15	15	15	16	16
9	16	16	16	17	17

Low Bin Capacity & High Number of Packages
With a bin capacity of 20 units, and with 100 packages. Blow is the graph of the genetic algorithms best fitness through generation counts.



Average Time(GA): 0.09226058006286621
Average Bin Count(GA): 30.52

Average Time(FF): 0.0005601310729980469
Average Bin Count(FF): 28.16

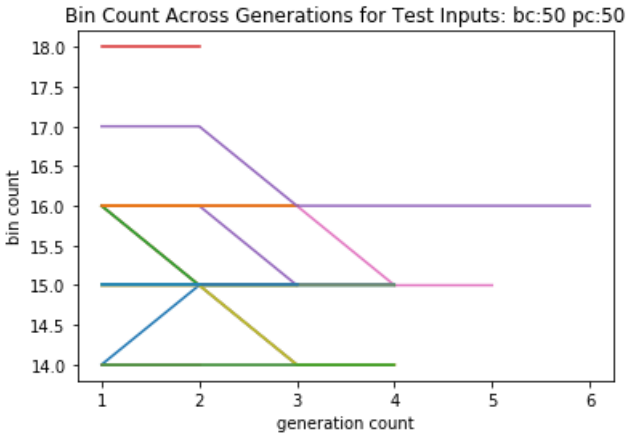
Average Time(BF): 0.0007603073120117188
Average Bin Count(BF): 29.88

Average Time(WF): Unmeasurable in Python
Average Bin Count(WF): 30.32

Average Time(NF): Unmeasurable in Python
Average Bin Count(NF): 30.32

	GA	FF	BF	WF	NF
0	30	28	28	30	30
1	29	28	28	30	29
2	32	29	29	31	32
3	31	29	29	30	30
4	30	28	28	31	31
5	32	29	29	30	31
6	32	30	30	32	33
7	29	27	27	28	28
8	31	29	28	29	31
9	28	26	26	28	28

High Bin Capacity & Low Number of Packages



Average Time(GA): 0.020444536209106447
Average Bin Count(GA): 15.16

Average Time(FF): 4.002571105957031e-05
Average Bin Count(FF): 14.48

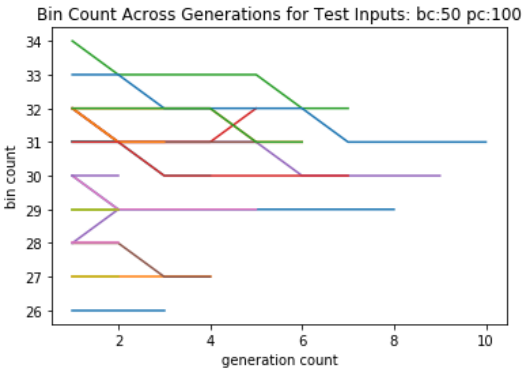
Average Time(BF): 0.00012002944946289062
Average Bin Count(BF): 15.24

Average Time(WF): 0.0
Average Bin Count(WF): 15.36

Average Time(NF): 0.0
Average Bin Count(NF): 15.36

	GA	FF	BF	WF	NF
0	15	15	15	15	16
1	15	15	15	15	16
2	15	15	14	16	16
3	16	15	15	17	16
4	16	14	14	15	15
5	15	14	14	14	14
6	15	15	15	16	15
7	15	15	15	15	16
8	15	14	14	15	15
9	15	14	14	15	15

High Bin Capacity & High Number of Packages



Average Time(GA): 0.10464350700378418
Average Bin Count(GA): 29.88

Average Time(FF): 0.00039998054504394534
Average Bin Count(FF): 27.44

Average Time(BF): 0.0003998279571533203
Average Bin Count(BF): 29.4

Average Time(WF): 0.00020003318786621094
Average Bin Count(WF): 29.76

Average Time(NF): 0.00020003318786621094
Average Bin Count(NF): 29.76

	GA	FF	BF	WF	NF
0	29	26	26	28	29
1	27	24	24	26	27
2	29	26	26	27	28
3	31	28	28	31	31
4	30	29	29	31	30
5	28	25	25	27	27
6	29	27	27	28	29
7	31	28	28	31	31
8	27	25	25	26	26
9	31	28	28	30	31

CONCLUSION

Permutations are a beneficial implementation of hybridization. The algorithm is a little slower averaging 2 nano seconds to 1 millisecond but has good solutions. Comparison to non-permutation-based implementations would be interesting considerations. Considering algorithms like next fit exists with absolute linear or absolute log linear big O, its likely this GA is near useless besides demonstration of ideology and permutation hybridization

REFERENCES

[1] Bin packing problem. (2020, October 13). Retrieved October 26, 2020, from https://en.wikipedia.org/wiki/Bin_packing_problem

[2] Jankovi, M. (2013, August 08). Genetic Algorithm for Bin Packing Problem. Retrieved October 26, 2020, from https://www.codeproject.com/Articles/633133/ga-bin-packing

[3] Michel Gourgand, Nathalie Grangeon, Nathalie Klement. An Analogy between Bin Packing Problem and Permutation Problem: A New Encoding Scheme. IFIP International Conference on Advances in Production Management Systems (APMS), Sep 2014, Ajaccio, France. pp.572-579, 10.1007/978-3-662-44739-0_70. hal-01388599