Packing Santa's Sleigh

Alexander Glandon, Bo Zhang, Rufor Chen, Vara Prasad Sheela
Dept. of Computer Science
Old Dominion University
Norfolk, Virginia, USA

Problem Description:

The goal of the Kaggle competition, "Packing Santa's Sleigh" is to arrange one million three dimensional (length*width*height) presents into a Sleigh of 1000*1000 (length*width) and infinite height. In evaluating a solution, the competition measures the total height of the present stack, and its order of presents. The side objectives are to sort the presents at the lowest total height and to maintain the original order of the presents as much as possible, such that lower present ID's remain at the top of the sleigh.

Methodology:

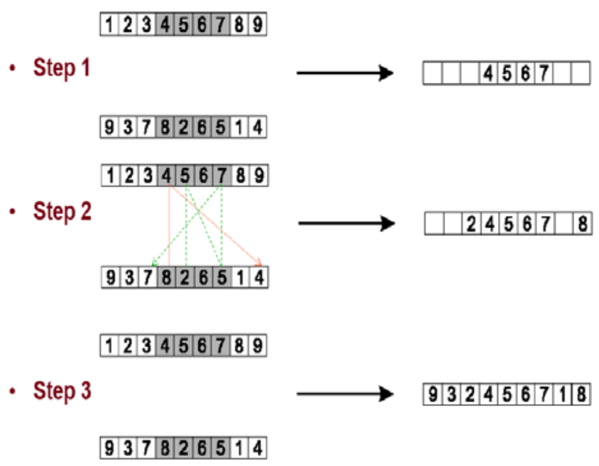
We implemented a genetic algorithm using C++ as the programming language. C++ was selected for its speed and thus potential for a higher "stop after no improvement" generation threshold. Each chromosome in the genetic algorithm represents a permutation of present ID placement order, each of which will generate a score using a placement heuristic that traverses this list.

Program design:

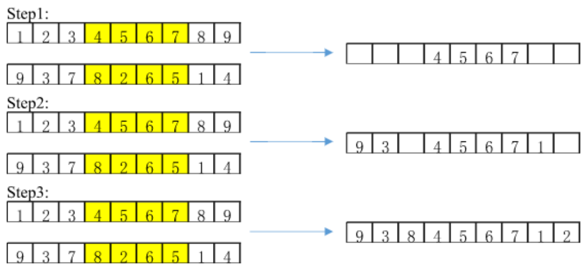
Chromosome Class: This consists of a permutation of one million presents and its corresponding performance score, generated by the placement heuristic in the Solution class.

Selection Function: This is used to randomly select the parent chromosomes for crossover, with some elitism; chromosomes with better performance are more likely to be chosen.

Crossover Function: We implemented a random selection between three standard crossover methods: Partially Mapped Crossover (PMX), Order 1 and Revised Partially Mapped Crossover (RPMX). RPMX is a non-standard method that is derived from PMX, which swaps the final two steps. The following figure represents an example of PMX, retrieved from the lecture slides:



The following figure represents an example of RPMX, with the same parent chromosomes:



Step 1: Copy crossover segment from Parent 1
Step 2: Copy values from Parent 2 not inside of the crossover segment index
Step 3: Copy values from Parent 2 inside of the crossover segment index

Mutation Function: We implemented a random selection between two standard mutation methods: Swap and Scramble. After some testing, we decided to exclusively use Swap to better preserve the present order when generating new chromosomes.

Solution Class: This contains the placement heuristic, which simulates the placement of a chromosome's present order into the container. The final version of the placement heuristic uses a layer-based approach to reduce execution time, which allows us to avoid iterating through the entire container for each present.

Issues & Optimization:

While building and testing the placement heuristic from the ground up, execution time and memory consumption became a serious problem with 1 million presents, and attempting to use the first iteration of the heuristic (which used a 3D array of integers, each number representing an empty space or present ID) from a scaled-down implementation of the problem with 100~1000 presents would not generate a score within a reasonable amount of time (>1 day).

To alleviate memory consumption, we abandoned the 3D array data structure in favor of exclusively using present corner coordinates to represent each present in the container in the same format of the submission file for the Kaggle competition, reducing the memory consumption significantly.

The superfluous execution time was due to redundant availability checking in our placement heuristic, since it brute-forced finding an empty space by iterating through the entire container for each present in a chromosome. To reduce this, we implemented a layer-based approach to finding empty space, which would sacrifice present compactness for speed, and is explained as follows:

For each candidate position, try a candidate rotation:

To determine a candidate position...

Fig 1: Order of searching the next position: left to right, top to bottom

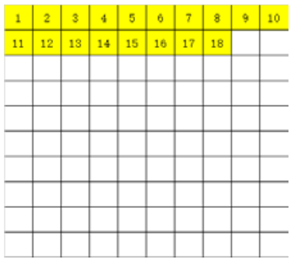


Fig. 2: If the present does not fit in the remaining horizontal space, look for the next empty row:

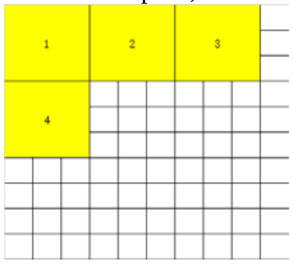


Fig. 3: If partially filled rows exist due to uneven present sizes, we still look for the next empty row instead:

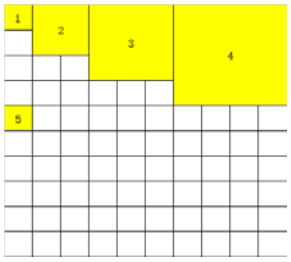
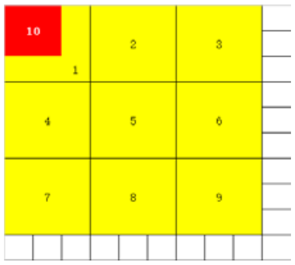
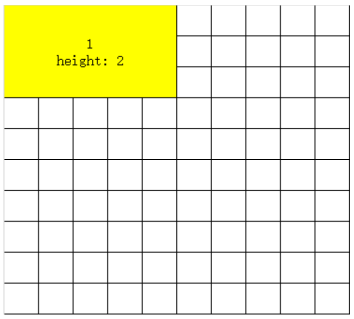


Fig. 4: If the present does not fit in the remaining horizontal and vertical space, we create a new layer and restart from the top left. This strategy is also used for z-layers in the final version, simulating "shelves"



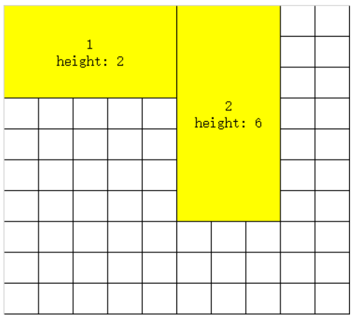
To determine a candidate rotation…

Fig. 5: For present 1 with dimensions (2,5,3), the lowest possible height is 2, and longest horizontal length is 5. It is placed in its first choice since there is enough empty space.



Present 1: (2,5,3)
Try height 2, length 5: Success

Fig. 6: For present 2 with dimensions (3,6,7), the lowest possible height is 3, and longest horizontal length is 7. It fails its first choice, and continues try a height of 3 and length of 6, then a height of 6 and length of 7, until finally it succeeds in fitting its fourth choice of a height of 6 and length of 3.



Present 2: (3,6,7)
Try height 3, length 7: Fail
Try height 3, length 6: Fail
Try height 6, length 7: Fail
Try height 6, length 3: Success

Alternative Design for Placement Heuristic:

Another placement heuristic was implemented with the strategy of grouping presents to preserve the original present order, and tracking empty space after placing each present. This puts more emphasis on present compactness instead of maintaining the original placement order. The steps to this heuristic is described as follows:

1) Shuffle the presents in groups of 10, with the highest presents in the first group of 10 and so on
2) Initialize the sleigh height as 0
3) While more presents remain to be placed
   a) Search for the smallest open space available that will fit the present
   b) If there are no open spaces that will fit the present:
      i) Create an open space on top of the sleigh filling the top 300 slices
      ii) place the present into this open space, and divide the remaining space into 3 or less open spaces and add them to the list of open spaces
   c) If there is an open space that will fit the present and it is in the middle of the sleigh
      i) Place the present into this open space, and divide the remaining space into 3 or less open spaces and add them to the list of open spaces
   d) If there is an open space that will fit the present and it is on the top of the sleigh
      i) Place the present into this open space, and divide the remaining space in the sleigh into 3 or less open spaces and add them to the list of open spaces, and divide the space over the top of the sleigh into 4 or less open spaces and add them to the list of open spaces also.

This heuristic is able to reduce the total height of candidate solutions more effectively, as it keeps track of every open space throughout the sleigh, but distributing packages over such a large space incurs a high penalty from the ordering section of the evaluation function. The other downside to this strategy was that it could not be implemented into the genetic algorithm approach, since it still took too long to traverse the container for each present.

Results & Conclusion:

As the total execution time of the second-to-last version of our program still ended up totalling somewhere between 3~30 hours, and we had multiple parameters to a genetic algorithm we intended to test (population, crossover probability, mutation probability), we were unable to create enough data samples to properly isolate each parameter at the time. Table 1 in the Appendix contains our test results for the second-to-last version of our program using a generation threshold of 5, and also contains our final version results (market with an asterisk "*") in which we adopt flat dividers between each layer, simulating shelves. This final version of our program was able to reduce the runtime to less than 10 minutes, and thus we were able to reproduce results for the parameters we tested in the previous version to confirm whether the trends we observed between our parameters and the performance score were consistent.

The dramatic improvement in performance score between the second-to-last and final version (~200-300 million) makes the impact of the ordering penalty much more obvious from the comparison to the results of our second-to-last version, which was able to keep the container more tightly packed at the cost of a high runtime. As shown in Chart 1 and Chart 2 in Appendix B, our original findings of a trend between population and performance still seem to hold to some extent, although the score difference is reduced to a negligible amount (<1000), rather than the several million that the second-to-last version indicated. Our best final score out of both versions remains with the parameter combination of 200 population, 60% crossover probability, and 100% mutation probability, with a best final score of 5,010,160 for the final version, and 289,720,668 for the second-to-last version.

Appendix A: Tables

Table 1, (Generation threshold of 5, “*” marks the entries from the final version):

Population Size	CrossOver/Probability	Mutation/Probability	Best Performance
*200	*Random/60	*Swap/100	*5010160
200	Random/60	Swap/100	289720668
*80	*Random/60	*Swap/100	*5010206
80	Random/60	Swap/100	292351202
*192	*Random/60	*Swap/100	*5010630
192	Random/60	Swap/100	292489062
*216	*Random/60	*Swap/100	*5010648
216	Random/60	Swap/100	294344408
*218	*Random/60	*Swap/100	*5010172
218	Random/60	Swap/100	295230058
*200	*Random/100	*Swap/100	*5010212
200	Random/100	Swap/100	295731320
*130	*Random/80	*Swap/100	*5010264
130	Random/80	Swap/10	295815360
*100	*Random/60	*Swap/100	*5010216
100	Random/60	Swap/100	296825288
*150	*Random/80	*Swap/10	*5010290

150	Random/80	Swap/10	297153446
*10	*Random/60	*Swap/100	*5023972
10	Random/60	Swap/100	298017374
*160	*Random/50	*Random/50	*5010296
160	Random/50	Random/50	298313676
*20	*Random/50	*Random/50	*5010638
20	Random/50	Random/50	299239456
*120	*Random/80	*Swap/10	*5010310
120	Random/80	Swap/10	299239456
*140	*Random/80	*Swap/10	*5010504
140	Random/80	Swap/10	299239456
*110	*PMX/60	*Swap/100	*5010218
110	PMX/60	Swap/100	299239456
*4	*Random/60	*Swap/10	*5011854
4	Random/60	Swap/10	307832492
*10	*PMX/80	*Swap/95	*5010848
10	PMX/80	Swap/95	312318490

Appendix B: Figures

Chart 1 (X-axis is Mutation Probability/Type, Crossover Probability/Type, Population; Y-axis is Performance)
Second-to-last version:

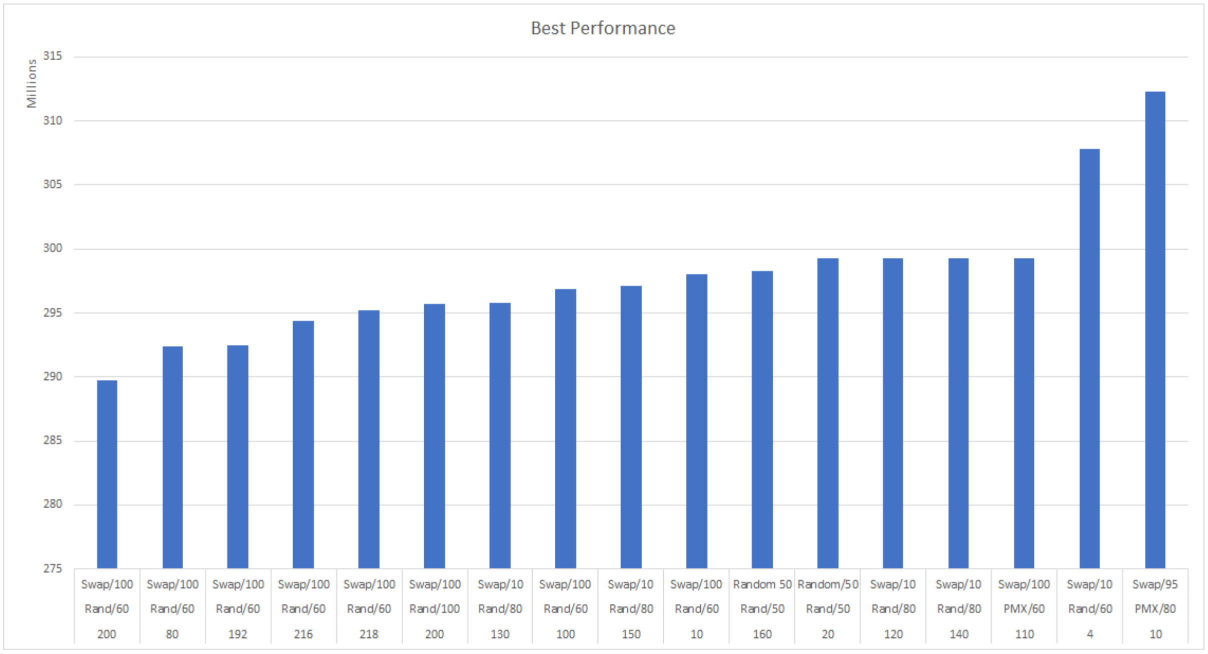
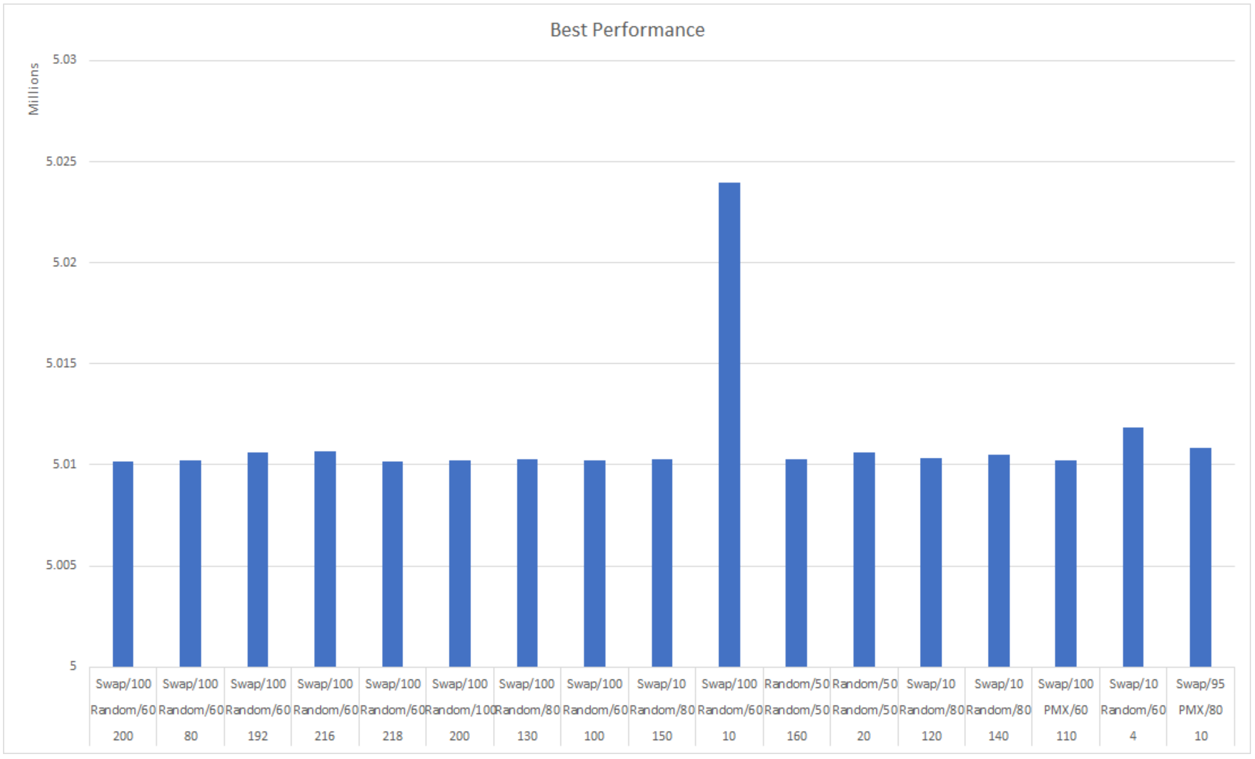


Chart 2 (X-axis is Mutation Probability/Type, Crossover Probability/Type, Population; Y-axis is Performance)
Final version:


Best Performance
Millions
Swap/100 Random/60 200
Swap/100 Random/60 80
Swap/100 Random/60 192
Swap/100 Random/60 216
Swap/100 Random/60 218
Swap/100 Random/100 200
Swap/100 Random/80 130
Swap/100 Random/60 100
Swap/10 Random/80 150
Swap/100 Random/60 10
Random/50 Random/50 160
Random/50 Random/50 20
Swap/10 Random/80 120
Swap/10 Random/80 140
Swap/100 PMX/60 110
Swap/10 Random/60 4
Swap/95 PMX/80 10