2nd Lt David Crow

ENG/20M

CSCE 686 Advanced Algorithms, Homework 5a

­

**Problem 1 – Talbi 1.8**

We consider in this exercise the bin packing problem. Let us consider an indirect encoding based on permutations. Propose a decoding function of these permutations that generates feasible solutions to the bin packing problem. This representation belongs to the one-to-many class of encodings. Analyze the redundancy of this encoding. How does the degree of redundancy grow with the number of bins?

We can imagine here that our decoder is simply an algorithm that outputs every possible solution for a valid input. In this way, our decoder is akin to an enumerator in computational theory [1]. In this problem, the input is some permutation of the items . By encoding our input in this way, we can easily iterate through all possible input permutations and search the entire search space (this search space, of course, is of size ) in an efficient manner [2].

Because we only care about a specific item’s *weight* (that is, the item’s name, volume, etc. do not matter to the algorithm), each can simply be a numeric representation of the item’s weight. Additionally, we can encode these items in any order (permutation). As an example, we can encode three items of weights one, two, and three in the following ways:

. Clearly, we’ve given all six valid permutations of such a set of items.

Encoding items in this way “belongs to the one-to-many class of encodings” because each encoding (that is, each permutation of ) can have more than one valid solution. Consider the set of three items previously listed. If our bin capacity , each of the six encodings gives four valid (optimal) solutions:

If we also consider suboptimal solutions, we have another three valid solutions, each of which places each item in a unique bin. Thus, our decoder can give seven valid solutions (four of which are optimal) for each permutation of .

Because we’ve defined our list as a set of numbers (either integer or real), our decoder can simply generate every permutation of the input (which is itself every permutation of ). For each permutation, we can apply a greedy algorithm like first fit or modified first fit decreasing (both are described in problem 2). Each solution given by the greedy algorithm is thus one solution our decoder should return. Of course, we should gather all solutions in a single structure and remove duplicates before returning the entire structure.

Clearly, redundant solutions are possible. We can have redundancies in the internal order of the items in a given bin and in the order of the bins themselves [3]. In the list above, 1 and 2 demonstrate the first type of redundancy (as do 3 and 4), and 1 and 3 (alternatively, 2 and 4) demonstrate the second type. Even for such a trivial example, we see that redundant solutions can be very common; we can reduce our solution set of four solutions to any single solution without losing our solution quality. In other words, encoding and decoding the items as described will give every valid solution, but it will also ensure we waste computation time and resources simply identifying duplicate solutions.

We can also see that redundancy increases as the optimal number of bins increases for some given input set . Naturally, a larger number of bins required means that we can fit fewer items – on average – into each bin. To achieve this effect for a given , we must reduce the bin capacity . This, in turn, means that each item takes up a larger portion of each bin. In the extreme case, we can see that for all is possible for some . In other words, each bin can only hold one item. In cases like these, we certainly reduce our intra-bin redundancies – there are fewer ways to rearrange the items in a bin if each bin holds fewer items – but our inter-bin redundancies increase [3]. Specifically, we can rearrange a relatively large number of bins in many more ways than we can a relatively small number of bins. For this reason, redundancy in decoding grows rapidly with increases in the optimal number of bins.

**References**

1. Sipser, M. Variants of Turing Machines. In *Introduction to the Theory of Computation, Third*

*Edition* (pp. 176-182). Boston, MA: Cengage Learning.

1. Talbi, E. Main Common Concepts for Metaheuristics. In *Metaheuristics: From Design to*

*Implementation* (pp. 34-48). Hoboken, NJ: John Wiley & Sons.

1. <https://core.ac.uk/download/pdf/52105416.pdf>

**Problem 2 – Talbi 1.13**

The bin packing problem is a well-known combinatorial problem with many applications, including container or pallet loading, loading trucks with weight capacities, and creating file backups in removable media. In this problem, objects of different weights must be packed into a finite number of bins (each of capacity , where is greater than the largest of the weights). The problem is to find the minimum number of bins into which the weights can be placed without violating the capacity of any bin. One example of a greedy algorithm for this problem is the first fit algorithm; this algorithm places each item into the first bin with a large enough remaining capacity to fit the item. This algorithm requires time. Propose some improvements for this greedy algorithm. *Hint: For example, one could sort the elements before packing the bins.*

In computational complexity theory, the bin packing problem (BPP) belongs to the NP-hard class of problems [1]. One of most trivial algorithms for solving BPP is the first fit (FF) algorithm, a greedy approach to BPP that traverses through the items in an arbitrary order and assigns each item to the first nonempty bin with enough remaining space for the item. If such a bin does not exist, the algorithm allocates a new, empty bin and places the item there. This is an

approximation algorithm. In other words, it’s not guaranteed to find the optimal solution for a given BPP instance. Specifically, FF approximates the optimal solution within a factor of two; FF returns bins as its solution, and in all cases [1]. As given in the problem statement, this algorithm runs in time.

There are multiple ways one can improve the FF algorithm. One such way entails sorting the items in decreasing order of weight before applying FF. We call this algorithm first fit decreasing (FFD), and, because the runtime is , the overall runtime is, in effect, no worse than for standard FF [2]. According to [3], FFD will return a solution number such that bins, which is always less than or equal to , the number returned by standard FF. Because FFD runs in the same time as FF, and because FFD can better approximate the optimal solution value , FFD is better than FF for all nontrivial problem instances. In fact, FFD is the de facto standard for solving BP [4].

Alternatively, we can sort the items in decreasing order of weight (as in FFD), and then allocate one bin to each item of weight . These are the first steps of the modified FFD (MFFD) algorithm, which is described in [5]. We can classify the remaining items as medium (), small ), and tiny ). BY selectively assigning these items to the *already open* bins (see [5] for the exact procedure), and by then applying FFD to any remaining items, we can achieve a number of bins , which is another improvement on our approximation [6]. Because each step in the algorithm is executed sequentially, and because the worst-case step requires , the runtime is again bound by . In other words, our MFFD solutions are closer to the optimal value, and they run in about the same time as FF.

Note that exact algorithms for BPP do exist. The MTP algorithm, named for Martello and Toth, is one such algorithm for BPP that will always return the optimal solution value [7]. Over the years, other researchers have made both speed and optimality improvements to MTP [1]. Of course, exact algorithms typically execute slower than do approximation algorithms, so one needing to find a solution for BP should always consider the time/optimality tradeoff.

**References**

1. <https://en.wikipedia.org/wiki/Bin_packing_problem>
2. <https://en.wikipedia.org/wiki/Sorting_algorithm>
3. <https://link.springer.com/chapter/10.1007/978-3-540-74450-4_1>
4. <https://pdfs.semanticscholar.org/fa6d/3585158f11d3b8ce16367b9894c80cdb53ac.pdf>
5. <https://en.wikipedia.org/wiki/Bin_packing_problem#First-fit_algorithm>
6. <https://link.springer.com/article/10.1007/BF02011198>
7. <http://www.or.deis.unibo.it/kp/Chapter8.pdf>