Recursive Balanced k-Subset Sum Partition for Rule-constrained Resource Allocation

Zhuo Li
The Hong Kong Polytechnic University, Hong Kong
cszhuoli@comp.polyu.edu.hk

Jiannong Cao
The Hong Kong Polytechnic University, Hong Kong
csjcao@comp.polyu.edu.hk

Zhongyu Yao
The Hong Kong Polytechnic University, Hong Kong
frank7.yao@connect.polyu.hk

Wengen Li
The Hong Kong Polytechnic University, Hong Kong
cswgli@comp.polyu.edu.hk

Yu Yang
The Hong Kong Polytechnic University, Hong Kong
csyyang@comp.polyu.edu.hk

Jia Wang
The Hong Kong Polytechnic University, Hong Kong
csjiawang@comp.polyu.edu.hk

ABSTRACT

Balanced rule-constrained resource allocation aims to evenly distribute tasks to different processors under allocation rule constraints. Conventional heuristic approach fails to achieve optimal solution while simple brute force method has the defect of high computational complexity. To address these limitations, we propose "*recursive balanced k-subset sum partition* (RBkSP)", in which iterative "cut-one-out" policy is employed that in each round, only one subset whose weight of tasks sums up to $1/k$ of the total weight of all tasks is taken out from the set. In a single partition, we first create a dynamic programming table with its elements recursively computed, then use "zig-zag search" method to explore the table, find out elements with optimal subset partition and assign different partitions to proper places. Next, to resolve conflicts during allocation, we use simple but effective heuristic method to adjust the allocation of tasks that is contradicted to allocation rules. Testing results show RBkSP can achieve more balanced results with lower computational complexity over classical benchmarks.

KEYWORDS

Balanced k-Subset Sum Partition, Rule-constrained Resource Allocation

ACM Reference Format:
Zhuo Li, Jiannong Cao, Zhongyu Yao, Wengen Li, Yu Yang, and Jia Wang. 2020. Recursive Balanced k-Subset Sum Partition, for Rule-constrained Resource Allocation. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3340531.3412076

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM '20, October 19–23, 2020, Virtual Event, Ireland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6859-9/20/10...$15.00
https://doi.org/10.1145/3340531.3412076

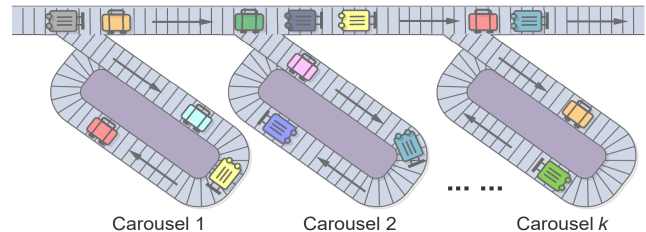


Figure 1: Illustration of carousel baggage allocation.

1 INTRODUCTION

In balanced rule-constrained resource allocation, it attempts to divide a set of tasks equally. Meanwhile, all allocations should strictly follow a series of predefined rules. Conventionally, heuristic approach (e.g., greedy algorithm) [2, 3] is used to sequentially distribute a set of jobs to whichever has the smallest sum of weights of tasks. It is intuitive but fails to obtain optimal allocation, because the performance of heuristic approach largely depends on the order of selected items. Besides, brute force [3, 5] is a straightforward way to solve this problem that lists all possible allocation of tasks, and find out the most balanced allocation plan fulling the allocation rules. Brute force can achieve optimal allocation but at the cost of very high computational complexity.

To address the aforementioned limitations, we propose "*recursive balanced k-subset sum partition* (RBkSP)" algorithm, in which "cut-one-out" policy is adopted that for the first partition, we only take one subset out, then repeat the process and take another subset out in the next partition. In each round, the key is to find out a partition with its weight sum equal to $1/k$ of the total weight of all elements in the set. Specifically, in RBkSP, we construct a table that stores *true*/*false* variables indicating the availability of the sum of all possible subsets. The value of elements in the table can be derived recursively using dynamic programming. After the table is filled up, "zig-zag search" is used to go back and forth across the table to search the optimal partition.

To verify RBkSP in real scenario, we instantiate it in airport carousel resource allocation since many airports worldwide still make baggage allocation plan manually. This lowers the operational efficiency as it unevenly distributes baggage to different carousels. We adopt a series of measures to make

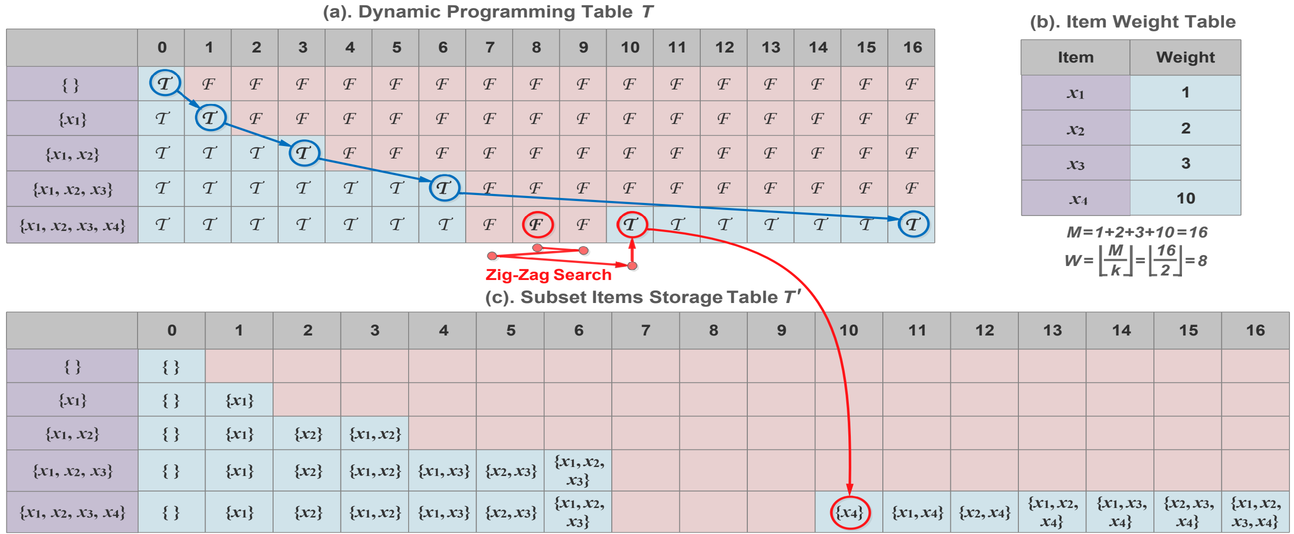


Figure 2: An example of finding a subset with a given sum by constructing dynamic programming table.

the allocation of baggages meet the allocation rules imposed by airport authority. Concretely, for each obtained partition of flights using RBkSP, we first select a flight with the highest priority and the largest weight as the leader, then assign all flights of this partition to the carousel that best suits the elected leader. Then, heuristic method is used to adjust flight-s among carousels according to their priorities and carried number of baggages so as to ensure that the allocation of all flights strictly follows allocation rules. Meanwhile, the load difference among carousels should be minimized as well.

2 PROBLEM FORMULATION

In this paper, we formulate balanced resource allocation as a k-subset sum partition problem. Given a set $\mathcal{S}$ of N tasks ($\mathcal{S} = \{x_1, ..., x_N\}$) with $w_1, ..., w_N$ the corresponding weights. Let $W = w_1 + ... + w_N$ the sum of weights of all elements, the goal is to find a partition scheme dividing $\mathcal{S}$ into k subsets, $\mathcal{S}_1, ..., \mathcal{S}_k$, whose subset sums, $W_1, ..., W_k$, are as close as $\overline{W}$, the arithmetic average of W ($\overline{W} = \frac{W}{k}$), i.e.,

$$\min \left| W_i - \overline{W} \right| = \min \left| \sum a_{i,j} w_j - \frac{W}{k} \right|, \tag{1}$$

where $i \in \{1, ..., k\}$ represents the i-th divided subset and $j \in \{1, ..., N\}$ denotes the index of elements. $a_{i,j} \in \{0, 1\}$ is a binary variable with 1 indicating item j is assigned to $\mathcal{S}_i$.

3 BALANCED *K*-SUBSET SUM PARTITION

Similar to but distinguished from 0/1 knapsack problem, the key to balanced k-subset sum partition, denoted by $\boldsymbol{P}(i, \mathcal{W})$, is to find a subset with a given sum, that is, pick a list of items whose sum of weights equals to $\mathcal{W}$:

$$\sum_{i=1}^{N} a_i w_i = \mathcal{W}, \; a_i \in \{0, 1\}. \tag{2}$$

where $a_i = 1$ means item x_i is picked and $a_i = 0$ otherwise.

There are k^N possible subsets for brute force search, and the increase of complexity would be of exponential order as the number of items N gets larger. Inspired by dynamic programming [1, 4], a better solution is to decompose this problem into a series of nested subproblems. In other words, considering the i-th item in $\mathcal{S}$, there are two possibilities associated with it, picking it or leaving it. On the one hand, if the i-th item is included in the optimal subset, the original problem is reduced to a subproblem that we need to find out a selection of items in the remaining $N-1$ items whose total weight is $\mathcal{W} - w_i$, i.e., $\boldsymbol{P}(i-1, \mathcal{W} - w_i)$. On the other hand, if it is not included in the optimal subset, we continue searching other items and make sure their total weight is equal to $\mathcal{W}$, i.e., $\boldsymbol{P}(i-1, \mathcal{W})$. Mathematically, the recurrence relation can be described by state transition function:

$$\boldsymbol{P}(i, \mathcal{W}) = \begin{cases} 0, \text{ if } i = 0; \\ 0, \text{ if } \mathcal{W} = 0; \\ \boldsymbol{P}(i-1, \mathcal{W}), \text{ if } w_i > \mathcal{W}; \\ \max\{\boldsymbol{P}(i-1, \mathcal{W}), \boldsymbol{P}(i-1, \mathcal{W} - w_i) + w_i\}, \text{ else.} \end{cases} \tag{3}$$

To solve Eq. (3), we first need to build up a dynamic programming table $\boldsymbol{T}$ of size $(N+1) \times (W+1)$, where N is the number of elements and $W = w_1 + \cdots + w_N$ is the sum of weights of all elements in $\mathcal{S}$ [1]. Figure 2 is a depiction of the table, in which the i-th row entry represents a subset $\mathcal{S}'_i$ of $\mathcal{S}$ containing the first i elements of $\mathcal{S}$, i.e., $\mathcal{S}'_i = \{x_1, x_2, ..., x_i\}$ $(0 \le i \le N)$, and the j-th column entry is an integer ranging from 0 to W $(0 \le j \le W)$. We set $\boldsymbol{T}[i][j] = true$ if the weights of a selection of items in $\mathcal{S}'_i$ sum up to j, and set $\boldsymbol{T}[i][j] = false$ otherwise. Meanwhile, we have to maintain another table $\boldsymbol{T}'$ of the same size as $\boldsymbol{T}$. In $\boldsymbol{T}'[i][j]$, it records the selected elements if $\boldsymbol{T}[i][j] = true$ or leaves it blank if $\boldsymbol{T}[i][j] = false$, thus we can easily identify these picked items if a subset with a given sum is found. Noted that when filling in the table $\boldsymbol{T}$, its element value can be derived according to Eq. (3) other than enumerating all possible combinations. Concretely, we

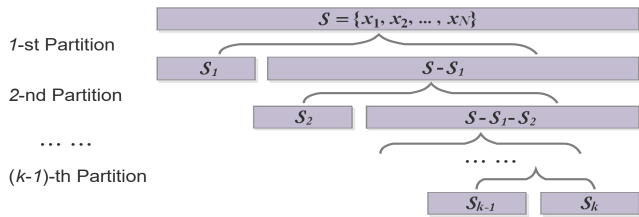


Figure 3: Illustration of the iterative "cut-one-out" policy in balanced k-subset sum partition.

initialize the first column to *true* and the first row except its first element to *false*, respectively. Then, the j-th element of the i-th row entry $\boldsymbol{T}[i][j]$ can be computed recursively, i.e., $\boldsymbol{T}[i][j] = (\boldsymbol{T}[i-1][j] \,||\, \boldsymbol{T}[i-1][j-w_i])$. The chain of blue circles and arrows in Figure 2 gives an intuitive example to demonstrate the recursive calculation process.

Subsequently, we set $j = \overline{W} = \lfloor \frac{W}{k} \rfloor$, then check the values on the bottom row of $\boldsymbol{T}$. If $\boldsymbol{T}[N+1][\overline{W}] = true$, it indicates we can find out a subset of $\mathcal{S}$ whose sum of weights is exactly equal to $\overline{W}$, i.e., the $1/k$ of total weight of all elements in $\mathcal{S}$. Then, the list of corresponding elements can be fetched in $\boldsymbol{T}'[N+1][\overline{W}]$. Moreover, we can find sometimes $\boldsymbol{T}[N+1][\overline{W}] = false$, which means the partition with exactly subset sum $\overline{W}$ cannot be obtained in some cases. Alternatively, we propose "zig-zag search" method to find the next best solution. More specifically, we start from checking $\boldsymbol{T}[N+1][\overline{W}+1]$ and we can get a successful subset partition if it is *true*, otherwise go back and check $\boldsymbol{T}[N+1][\overline{W}-1]$, then continue checking $\boldsymbol{T}[N+1][\overline{W}+2]$, $\boldsymbol{T}[N+1][\overline{W}-2]$, and so forth. Searching is conducted back and forth until it triggers the threshold, i.e., $\overline{W} - L \leq \overline{W} \pm l \leq \overline{W} + L$. Figure 2 gives an intuitive example that $\mathcal{S} = \{x_1, x_2, x_3, x_4\}$ with corresponding weights $w_1, w_2, w_3, w_4 = 1, 2, 3, 10$, respectively, and let $k = 2$, we have $N+1 = 5$, $W = 16$ and $\overline{W} = \lfloor \frac{W}{k} \rfloor = 8$. As we can see $\boldsymbol{T}[5][8] = false$, then move to check $\boldsymbol{T}[5][9]$, $\boldsymbol{T}[5][7]$ until $\boldsymbol{T}[5][10] = true$ is found. Accordingly, an optimal subset $\{x_4\} \in \mathcal{S}$ is obtained through looking up the table $\boldsymbol{T}'$.

So far, we have successfully made the 1st partition on $\mathcal{S}$ and obtain subset $\mathcal{S}_1$, then without loss of generality, let's extend it to $k \geq 3$, we would love to take the 2nd subset $\mathcal{S}_2$ out from the remaining part $\mathcal{S} - \mathcal{S}_1$. In this manner, in each round, we pick up some items from the remaining part of previous partition whose weight sum is equal to $1/k$ of the total weight of all elements in $\mathcal{S}$, then repeat this in the next round. It is what we call "cut-one-out" policy that $k-1$ iterations will be conducted if we want to divide a set into k piles, and the process of it is illustrated by Figure 3.

4 APPLICATION IN RULE-BASED RESOURCE ALLOCATION

We evaluate RBkSP on airport carousel resource allocation, which, in real operation, is constrained by a group of allocation rules set by airport authority. Table 1 lists a part of high priority rules in daily airport baggage allocation.

Having all arriving flights during a short period of time (usually 1 hour) divided into k groups, the next step is to assign them to different carousels. Briefly, we first give a tag to each flight containing the information of priority p and weight w (number of baggages). They are organized in the form of (*flight no.*, *priority*, *weight*), say, (SQ856, p:1, w:450). For each group, the flight with the highest priority and largest weight is elected as the group leader, then all flights in this group will be assigned to a carousel that the group leader best suits for. But this has not finished yet, because the allocation of some flights may violate allocation rules, for example, two heavy loading flights are assigned to the same carousel. However, these allocation rules are trivial, it is challenging to mathematically formulate it and find the optimal solution. Alternatively, we use heuristic method to adjust the allocation of flights which do not satisfy the rules. Empirically, if two flights assigned to the same carousel are conflict with each other, we remove the one with lower priority, and switch it with a flight allocated to other carousels. In order to keep load balanced across different carousels, the number of baggages carried by two switching flights should be closed to each other. Figure 4 gives an illustrative example to help readers better understand the process how we resolve allocation conflicts using heuristic approach.

Priority	Specification
Rule 1	A380 should be allocated to Carousel 10 & 12.
Rule 2	KA & CX flights should not be assigned to Carousel 2, 3, 5 & 6.
Rule 3	Heavy loading flights shall not be assigned on the same arrival carousel.
Rule 4	All KA flights should not be allocated with EK flights on the same arrival carousel.
… …	

Table 1: Rule specifications on flight allocation (KA, CX and EK, etc., refer to airline code).

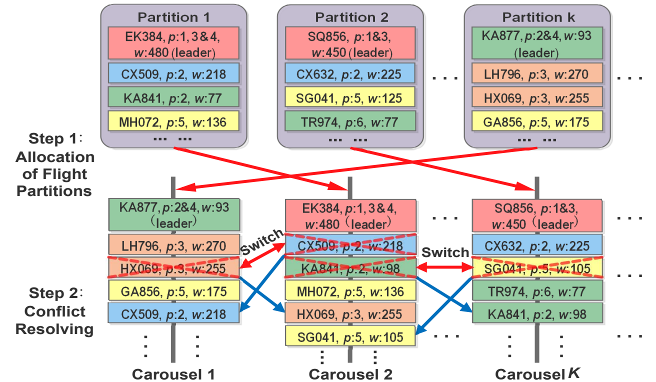


Figure 4: Illustration of resolving allocation conflicts among flights using heuristic approach.

5 EXPERIMENT

5.1 Evaluation Metric

We use the mean absolute error (MAE) to evaluate the performance of the proposed algorithm:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{k} \sum_{i=1}^{k} \left| W_i - \frac{W}{k} \right|. \quad (4)$$

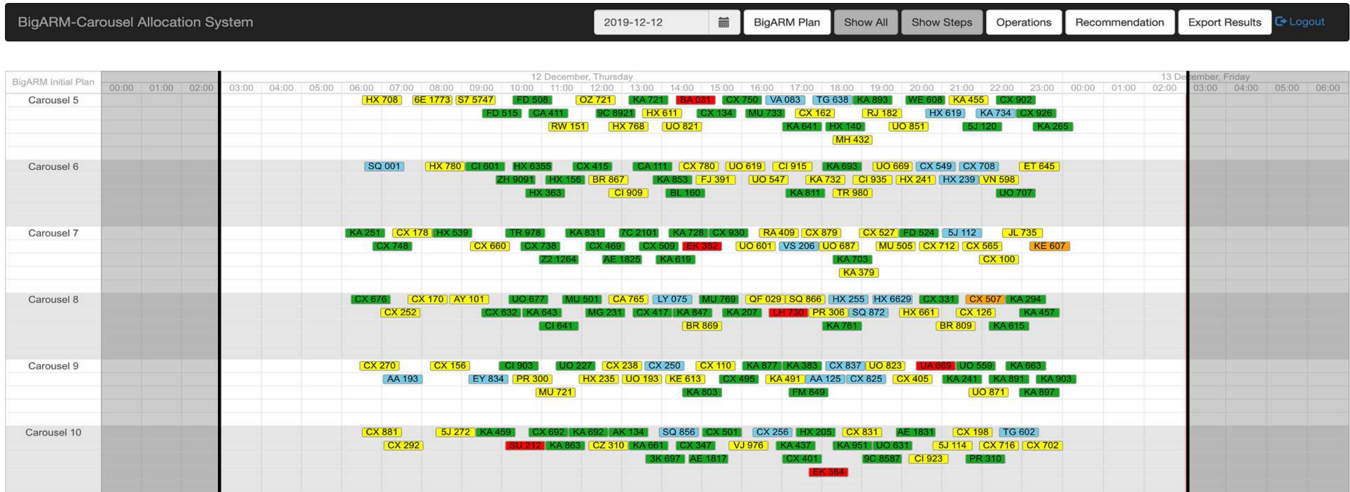
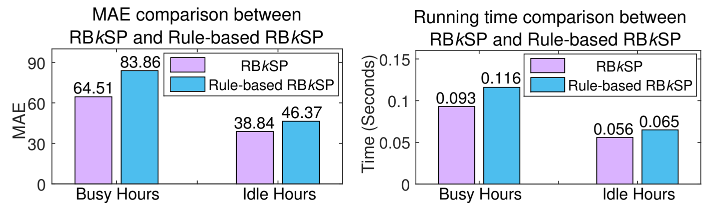


Figure 5: Demonstration of the developed carousel resource allocation system based on RBkSP.

Models	MAE		Running Time	
	Busy	Idle	Busy	Idle
Greedy	175.32	65.45	0.052 s	0.039 s
Brute Force	64.51	38.84	5.854 s	1.873 s
RBkSP	64.51	38.84	0.093 s	0.056 s

Table 2: Comparison between models.



(a) MAE Comparison. **(b) Running Time Comparison (s).**

Figure 6: Compare RBkSP with Rule-based RBkSP.

5.2 Dataset

The dataset we use is provided by a third party cooperator for scientific research purpose only. It includes 3-year's flight information of a busy international airport in Asia, including flight no., carried number of baggages, weather data, etc.

5.3 Experimental Results

5.3.1 Demonstration. Figure 5 demonstrates the carousel resource allocation system equipped with RBkSP developed by us. It presents the allocation plan of flights at arrival hall, which has 6 carousels indexed from 5 to 10. Red and orange represent heavy loading flights, and yellow and green indicate light loading flights. We can observe that arriving flights are evenly distributed to different carousels in time order.

5.3.2 Comparison between Models. Table 2 compares MAE across different models. RBkSP achieves lower MAE than greedy algorithm [2, 3]. This is because RBkSP can obtain optimal partition, but greedy is a heuristic algorithm that assigns items in a set sequentially to whichever subset has the smaller sum. In idle hours, the MAE difference between them is much smaller because carousel resource is relatively adequate during idle hours. Brute force [3, 5] can also make optimal partition since it enumerates all subsets and pick k subsets that have the smallest subset sum differences, but it is at the cost of much higher computational time.

5.3.3 Comparison with Rule-based RBkSP. As we can see in Figure 6(a), when applying RBkSP to carousel baggage allocation under some rule constraints, the MAE of rule-based RBkSP worsens, particularly during busy hours, because it has to spend more time on adjusting the allocation of flights across different carousels such that all allocation rules can be satisfied. However, in idle hours, the performance deteriorates a little bit as the number of flight adjustments is much less.

6 CONCLUSION

In this paper, we propose "*recursive balanced k-subset sum partition* (RBkSP)" with the aim to divide a set of tasks into k subsets with equal subset sum. It is successfully applied to rule-constrained airport resource allocation, and achieves more balanced allocation and comparably less running time.

7 ACKNOWLEDGEMENT

This work was supported by Hong Kong Innovation and Technology Fund (ITF) (ITC No. ITP/024/18LP) and Guangdong Key Area R&D Plan (Project No. 2020B010164002).

REFERENCES

[1] R. Bellman. 2003. *Dynamic Programming.*
[2] P. E Black. 2005. *Greedy Algorithm.* Technical Report. Dictionary of Algorithms and Data Structures.
[3] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. 2001. *Introduction to Algorithms.*
[4] R. E. Korf. 1998. A Complete Anytime Algorithm for Number Partitioning.
[5] R. E. Korf. 2009. Multi-Way Number Partitioning.