Solving Billion-Scale Knapsack Problems
解决十亿规模的背包问题

Xingwen Zhang, Feng Qi, Zhigang Hua and Shuang Yang
Ant Financial Services Group, San Mateo, CA 94402
{xingwen.zhang,feng.qi,z.hua,shuang.yang}@antfin.com

ABSTRACT

Knapsack problems (KPs) are common in industry, but solving KPs is known to be NP-hard and has been tractable only at a relatively small scale. This paper examines KPs in a slightly generalized form and shows that they can be solved *nearly optimally* at scale via distributed algorithms. The proposed approach can be implemented fairly easily with off-the-shelf distributed computing frameworks (e.g. MPI, Hadoop, Spark). As an example, our implementation leads to one of the most efficient KP solvers known to date – capable to solve KPs at an unprecedented scale (e.g., KPs with 1 billion decision variables and 1 billion constraints can be solved within 1 hour). The system has been deployed to production and called on a daily basis, yielding significant business impacts at Ant Financial.

CCS CONCEPTS

• **Information systems → Information systems applications**.

KEYWORDS

Knapsack Problems; Large-scale Optimization; Distributed Algorithms; MapReduce; Dual Descent; Synchronous Coordinate Descent

1 INTRODUCTION

Knapsack problems (KPs) [13] are commonly seen in real-world applications, for example, budget allocation and pacing (as in advertising and marketing), online traffic control (as in search engine and recommender systems), logistics optimization (as in e-commerce), asset management (as in finance) and so on. Unfortunately, solving KPs is well known to be NP-hard and in practice only feasible at a relatively small scale even with commercial solvers [1, 5, 9–11, 21].

We are primarily motivated by KPs as emerged in an internet industry setting, where decisions need to be made on a per user basis while the number of users can be large (e.g., billions). The "resources" to be allocated in a KP can be either financial (e.g., loans, marketing promotions, ads spending, asset portfolios) or non-monetary (such as user impressions, clicks, dwell time). Typically,

we want to optimize an *objective* (e.g., the expected user conversions as in the case of marketing campaign) subject to a set of constraints, which can be roughly divided into two types: *global constraints* that often limit the maximum allowance of the resources at a global level, and *local constraints* that impose further restrictions specific to individual users / user groups. In practise, while the size of the global constraints are often small (e.g. a few hundreds), the typical scales for both the decision variables and the local constraints can be at the level of billions. Unfortunately, solving KPs at such scales has been an open technical challenge [1, 5, 9–11, 21].

We present in this paper one of the first attempts to solve real-world KPs at billion-scale. Firstly, using the *MapReduce* computing model, we design a distributed framework for solving KPs by exploiting the decomposability of the dual problems and *dual descent* (DD). Secondly, to further improve convergence especially as the DD algorithm is prone to constraint violations when implemented in a distributed setting, we developed the *synchronous coordinate descent* (SCD) algorithm that doesn't suffer from these issues. Furthermore, by leveraging the hierarchical structures of the local constraints, we show that the integer programming sub-problem can be solved *optimally in polynomial time by a greedy algorithm*, dramatically improving the quality and solving speed for KPs with hierarchical local constraints. Lastly, we implement our algorithm with off-the-shelf distributed computing frameworks (e.g. MPI, Hadoop, Spark), leading to one of the most efficient KP solvers known to date (e.g., KPs with 1 billion decision variables and 1 billion constraints can be solved within 1 hour). Our work also contributes to a deployed system that's being used for production decision making at Ant Financial everyday.

2 PROBLEM FORMULATION

Consider the following *generalized* variant of the knapsack problem (1)–(4), where a set of M items are to be allocated among a set of N users, subject to K global constraints and L local constraints. The global constraints (2) limit the resource allocation for each *knapsack*, whereas the local constraints (3) restrict per-user consumption. If item j is allocated to group[1] i, i.e. $x_{i,j} = 1$, we gain a profit of $p_{i,j}$ and consume $b_{i,j,k}$ amount of resources for the k-th knapsack ($k \in [K] = \{1, \ldots, K\}$). B_k and C_l are *strictly positive* while $p_{i,j}$ and $b_{i,j,k}$ are *non-negative*.

---

[1]Throughout this paper, the terms "user" and "group" are used interchangeably, as a "user" in our setting corresponds to a "group" in the *operations research* literature.

$$\max_{x_{i,j}} \sum_{i=1}^{N} \sum_{j=1}^{M} p_{i,j} x_{i,j} \tag{1}$$

$$s.t. \sum_{i=1}^{N} \sum_{j=1}^{M} b_{i,j,k} x_{i,j} \leq B_k, \ \forall k \in [K] \tag{2}$$

$$\sum_{j \in S_l} x_{i,j} \leq C_l, \ \forall i \in [N], S_l \subseteq [M], \forall l \in [L] \tag{3}$$

$$x_{i,j} \in \{0,1\}, \forall i \in [N], \forall j \in [M]. \tag{4}$$

Note that although we assume x_{ij}'s are *binary* (i.e., $x_{ij} \in \{0,1\}$), our approach is equally applicable to *categorical* (i.e., non-negative integer) variables. In fact, our implementation supports both binary and categorical decision variables.

2.1 Hierarchical Local Constraints

We further explore a more complex case of (1)–(4), where there are hierarchical structures in the local constraints (3) such that the index sets are *either* disjoint *or otherwise* nested. Formally,

Definition 2.1. Local constraints (3) are said to be *hierarchical* when the following holds: $\forall l, l' \in [L]$, if $S_l \cap S_{l'} \neq \emptyset$, then *either* $S_l \subseteq S_{l'}$ *or* $S_{l'} \subseteq S_l$.

This property is commonly seen in real-world, where items are not independent from each other but rather related and often the time organized as nested groups (e.g., according to a taxonomy). A *directed acyclic graph* (DAG) can be constructed for $\{S_l | l \in [L]\}$, i.e., an arc from S_l to $S_{l'}$ iff $S_l \subseteq S_{l'}$. In the trivial case where items are unrelated, this DAG will degenerate to a *set* where S_l's are disjoint from each other.

2.2 Connections to Other KP Variants

A few variants of the knapsack problems have been studied in the literature, including *multi-dimensional knapsack problems* (MD-KPs) [3], *multi-choice knapsack problems* (MCKPs) [13], and *multi-dimensional multi-choice knapsack problems* (MMKPs) [14]. An MDKP has a single item and multiple knapsack constraints, when the item is chosen, resources from multiple knapsacks will be consumed. MCKP is an extension of the classical single-constraint KP, where the items are partitioned into multiple groups and exactly one item from each group can be chosen. MMKP is a combination of MDKP and MCKP [12, 14].

The problems we study here (1)–(4), including the case with the local hierarchical assumption, are more generalized compared to the other variants as they allow more flexible forms of constraints. In fact, *all* these classical variants can be seen as special cases of our formulation. For example, if there's only one item and no local constraint (i.e., $M = 1$ and $L = 0$), it reduces to MDKPs [3]; when $K = 1, C. = 1$ and $L = 1$, it becomes MCKPs [13]; and when $C. = 1$ and $L = 1$, MMKPs [14].

3 RELATED WORK

As a famous example of the *integer programming* (IP) problem, KPs (including vanilla KP and its variants) are well-known to be NP-hard [13, 14, 17]. Both exact and heuristic algorithms have been studied

for solving these problems in operations research (OR), including, for example, *branch and bound* [3], *tabu search* [8], *simulated annealing search* [15] and so on. See [16] for a recent survey on solving MDKPs. Unfortunately, these traditional OR algorithms were not designed for modern infrastructure, in particular distributed computing. As a result, they can only solve KPs at a very limited scale (i.e., thousands to millions of decision variables).

The recent works by Pinterest [21] and LinkedIn [9, 10] are the only few cases that have examined KPs at a scale close to ours. In [21], a simplified KP is solved to decide notification volume for each user so as to optimize long-term user engagements. Their *threshold search* algorithm is able to solve KPs at the scale of hundreds of millions, but only when there is a single global constraint. In [9, 10], the authors formulated the email volume optimization problem at LinkedIn as a multi-objective *linear programming* (LP) problem, which is converted to a strongly convex dual problem with an added quadratic regularization term as in [1], and the set of dual multipliers is used for online production serving. To make the solution tractable, clustering or sampling techniques are used to reduce the number of decision variables [1]. The LP relaxation as well as the downsizing procedure, however, inevitably hurt the quality of the resultant solution (i.e., optimality and constraint satisfaction).

4 OUR APPROACH

The scale of the problems we aim to solve suggests that the solution ought to be computed by a distributed cluster rather than a single machine. In this section, we first introduce the dual decomposition of KPs that opens the door to distributed solving, and then develop distributed solving algorithms described using the *MapReduce* computing model[2] [7].

4.1 Dual Decomposition for KPs

Let us examine the dual problem of (1)–(4). By introducing a set of Lagrangian multipliers[3] $\{\lambda_k | k \in [K]\}$ (i.e., one for each global constraint), we have

$$\max_{x_{i,j}} \sum_{i=1}^{N} \sum_{j=1}^{M} p_{i,j} x_{i,j} - \sum_{k=1}^{K} \lambda_k (\sum_{i=1}^{N} \sum_{j=1}^{M} b_{i,j,k} x_{i,j} - B_k) \tag{5}$$

$$s.t. \sum_{j \in S_l} x_{i,j} \leq C_l, \ \forall i \in [N], S_l \subseteq [M], \forall l \in [L] \tag{6}$$

$$x_{i,j} \in \{0,1\}, \forall i \in [N], \forall j \in [M] \tag{7}$$

with optimality conditions

$$\lambda_k (\sum_{i=1}^{N} \sum_{j=1}^{M} b_{i,j,k} x_{i,j} - B_k) = 0, \forall k \in [K] \tag{8}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M} b_{i,j,k} x_{i,j} - B_k \leq 0, \forall k \in [K] \tag{9}$$

$$\lambda_k \geq 0, \forall k \in [K]. \tag{10}$$

---

[2]The MapReduce semantics are used here only to describe the algorithm. The implementation, however, can be based on any other distributed computing frameworks (e.g., MPI, Hadoop, Spark).
[3]In economics, λ_k is often interpreted as the *shadow price* (or *marginal utility*) of the k-th knapsack resource [2].

The maximization problem in (5) can be decomposed into a set of subproblems (i.e., one for each group i),

$$\max_{x_{i,j}} \sum_{j=1}^{M} p_{i,j} x_{i,j} - \sum_{k=1}^{K} \lambda_k \sum_{j=1}^{M} b_{i,j,k} x_{i,j} \qquad (11)$$

$$s.t. \sum_{j \in S_l} x_{i,j} \leq C_l, S_l \subseteq [M], \; \forall l \in [L] \qquad (12)$$

$$x_{i,j} \in \{0, 1\}, \forall j \in [M]. \qquad (13)$$

Given λs, these subproblems are independent from each other. This nice decomposability of the dual suggests a natural decentralized approach for solving large-scale KPs, i.e., by alternating between (1) solving the IP subproblems *in parallel* with given λ; and (2) updating λ while fixing the solution x.

4.2 Solving the IP Subproblems

Thanks to the decomposability of the dual, at a fixed λ, KPs can be solved at the individual user level *independently in parallel*. Compared to the original KPs, the per-user IP subproblems (11)-(13) are much easier to solve as (1) the scale is tiny – there're only $O(M)$ decision variables (*vs $O(MN)$* of the original KPs); and (2) the problem is much simpler (e.g., there's no global constraint). In practice, it's straightforward to solve these subproblems, e.g., by bundling an off-the-shelf solving subroutine (e.g., [5, 11, 18]) into the deployable image of the mapper.

For the more complex case with hierarchical local constraints, there's a fast algorithm that has a polynomial time complexity and is provably optimal. In particular, we design a greedy algorithm, as described in Algorithm 1, which solves the IP subproblems by traversing the DAG in a topological order. The algorithm orders items in a non-decreasing order of *cost-adjusted profit* (which is also the contributing dual value of $x_{i,j}$),

$$\tilde{p}_{i,j} = p_{i,j} - \sum_{k=1}^{K} \lambda_k b_{i,j,k}.$$

1

Starting from the items at the lowest level of the DAG, for each S_l the algorithm chooses its items in a non-decreasing order of the cost-adjusted profit until their sum exceeds C_l. The $x_{i,j}$ for the unchosen items in S_l are all assigned with value 0 and these items will not be considered in the subsequent iterations. This greedy selection process is repeated until all the nodes in the DAG have been traversed.

Initialize for $j \in [M]$: $x_{i,j} = 1$ if $\tilde{p}_{i,j} > 0$, or 0 otherwise
Sort $\{j\}$ in non-increasing order of $\tilde{p}_{i,j}$
for S_l in the topological order of the DAG **do**
    fetch the indices $j \in S_l$ with $x_{i,j} = 1$
    update $x_{i,j} = 0$ if j is not in top C_l sorted indices
end
return $\{x_{i,j}|j \in [M]\}$
Algorithm 1: Greedy algorithm for solving the per-user IP subproblem (11)–(13) with local hierarchical constraints.

This greedy algorithm has a polynomial time complexity, and as is shown later it's orders of magnitude faster than competitive solvers. We also prove it is optimal.

PROPOSITION 4.1. *Algorithm 1 optimally solves the integer programming problem (11)–(13) with hierarchical local constraints.*

PROOF. Given any other solution (denoted by $\{\tilde{x}_{i,j}, j \in [M]\}$) that satisfies the constraints (12) and (13) but differs from the greedy solution (denoted by $\{x^*_{i,j}, j \in [M]\}$), we can identify the first node in the topological order of the DAG at which the items chosen are different. Due to the nature of the greedy algorithm, there must exist a pair of items j and j' at the node where the adjusted profit of item j is no less than that of item j', but $x^*_{i,j} = 1, x^*_{i,j'} = 0, \tilde{x}_{i,j} = 0, \tilde{x}_{i,j'} = 1$. We can modify $\tilde{x}$ by setting $\tilde{x}_{i,j} = 1, \tilde{x}_{i,j'} = 0$ without decreasing the objective value of (11). All the constraints (12) and (13) are still satisfied, because any later node in the topological order of the DAG contains both j and j' or neither. In this way, we can always convert any solution to the greedy solution without decreasing the objective value or violating any constraint. This completes the proof. □

4.3 Distributed Algorithms for Solving KPs

4.3.1 Dual Descent. Our first algorithm for solving KPs is distributed *dual descent* (DD). It's an iterative procedure developed based on the decomposability of the dual. Given the solutions of the per-user subproblems, $\{x_{ij}\}$, the multipliers λ can be updated:

$$\lambda_k^{t+1} = \max(\lambda_k^t + \alpha(\sum_{i=1}^{N} \sum_{j=1}^{M} b_{i,j,k} x_{i,j} - B_k), 0), \qquad (14)$$

where the hyper-parameter α is the learning rate. In Algorithm 2, we detail it using the MapReduce model [7]. Particularly, in each iteration: 1) first, the solutions $x_{i,j}$ for the subproblems are computed independently in mappers (e.g., by off-the-shelf IP solvers [5, 11, 18], or in the hierarchical case by our greedy algorithm); once finished, each mapper emits K values $\{v_{ik} = \sum_{j=1}^{M} b_{i,j,k} x_{i,j} | k \in [K]\}$ corresponding to group i's resource consumption from each of the K knapsacks; 2) then, the reducers aggregate the total resource consumption of each knapsack, $R_k = \sum_{i=1}^{N} v_{ik}$; and 3) finally, a master node updates λ_k using dual descent (14).

4.3.2 Synchronous Coordinate Descent. This DD algorithm, however, is problematic because: (1) it requires a hyper-parameter α that needs to be chosen either manually or programmatically, which can be practically cumbersome or computationally intensive especially for large-scale KPs; and (2) as we show empirically, DD's very prone to constraint violations and as a result the resultant solutions are often invalid. To this end, we propose to use *coordinate descent* (CD), which updated λ_k one coordinate at a time while keeping other coordinates $\{\lambda_{k'}|k' \neq k\}$ fixed.

The CD algorithm can be further enhanced for the hierarchical case. Specifically, we note that it's possible to dramatically reduce the search space for λ by only considering a small set of candidate values. To show this, observe from Algorithm 1 that the solution only depends on the relative order of the cost-adjusted profits, $\tilde{p}_{i,j}$, rather than their actual values. For any given i, let $z_{j,k} = (p_{i,j} - \sum_{k'=1, k' \neq k}^{K} \lambda_{k'} b_{i,j,k'}) - \lambda_k b_{i,j,k}$ denote a linear function $z_{j,k}(\lambda_k)$, which defines a straight linear in the $(\lambda_k, z_{j,k})$-axis. The relative order of $\tilde{p}_{i,j}$ can only possibly change *either* at (1) the pairwise intersections of these M lines (i.e., $\{\lambda_k | z_{j,k}(\lambda_k) =$

Procedure Map(p_i, b_i, λ^t)
    solve $x_{i,j}$ from (11)
    for each $k \in \{1, \ldots, K\}$ **do**
        emit($k, v_{ik} = \sum_{j=1}^{M} b_{i,j,k} x_{i,j}$)
    end

Procedure Reduce(k, v_{ik})
    return $\sum_i v_{ik}$

initialize λ^0
for $t \leftarrow 0$ **to** $T - 1$ **do**
    for each $i \in \{1, \ldots, N\}$ in parallel **do**
        Map(p_i, b_i, λ^t)
    end
    for each $k \in \{1, \ldots, K\}$ in parallel **do**
        $R_k = $ Reduce(k, v_{ik})
    end
    for each $k \in \{1, \ldots, K\}$ **do**
        $\lambda_k^{t+1} = max(0, \lambda_k^t + \alpha(R_k - B_k))$
    end
    if λ^t has converged **then return** λ^t
end
return λ^T

Algorithm 2: Distributed dual descent algorithm for solving KPs.

$z_{j'\neq j,k}(\lambda_k), \forall j, j' \in [M]\}$), or at (2) their intersections with the horizontal axis (i.e., $\{\lambda_k | z_{j,k}(\lambda_k) = 0, \forall j \in [M]\}$). Therefore, it suffices to only screen λ_k at these values instead of over the entire interval $[0, \infty)$. Algorithm 3 describes this procedure for deciding potential candidate values for λ_k.

Procedure CalculateIntersectionPoints(p_i, b_i, λ^t, k)
    $\Lambda_k = \emptyset$
    for each $j \in \{1, \ldots, M\}$ **do**
        $\Lambda_k = \Lambda_k \cup \{\lambda_k | z_{j,k}(\lambda_k) = 0\}$
        for each $j' \in \{j + 1, \ldots, M\}$ **do**
            $\Lambda_k = \Lambda_k \cup \{\lambda_k | z_{j,k}(\lambda_k) = z_{j',k}(\lambda_k)\}$
        end
    end
    return the set of unique values in Λ_k

Algorithm 3: Calculating candidate values for λ_k (in parallel for each group i) for KPs with hierarchical local constraints.

Once we have the candidate λ values, for each group i *independently in parallel*, we can update λ using *synchronous coordinate descent* (SCD), as described in Algorithm 4. Specifically, for each group i, the mapper goes through the candidate values for λ_k and calculates the amount of k-th knapsack resource that would be used if we update λ_k to the corresponding candidate value. The mapper sorts the candidates of new λ_k in a decreasing order and emits only the *incremental* change as we decrease λ_k. For each global constraint k, the reducer then aggregates the emitted results for each key, and then λ_k is updated to the minimal threshold such that the total resource used does not exceed B_k.

While we employ synchronous CD that updates λ_k for all $k \in [K]$ simultaneously, other variants of CD, such as *cyclic* CD (updates one multiplier at a time) and *block* CD (updates multiple multipliers in parallel) are also applicable [20]. In our implementation, all these modes are supported, although SCD turns to perform the best.

Procedure Map(p_i, b_i, λ^t)
    for each $k \in \{1, \ldots, K\}$ **do**
        $\Lambda_k = $ CalculateIntersectionPoints(p_i, b_i, λ^t, k)
        Sort Λ_k in a decreasing order
        previous_sum = 0
        for each λ_k in Λ_k **do**
            $\lambda = (\lambda_1^t, \ldots, \lambda_{k-1}^t, \lambda_k, \lambda_{k+1}^t, \ldots, \lambda_K^t)$
            solve $x_{i,j}$ with multipliers λ using Algorithm 1
            current_sum = $\sum_{j=1}^{M} b_{i,j,k} x_{i,j}$
            if current_sum > previous_sum **then**
                $v_1 = \lambda_k$
                $v_2 = $ current_sum - previous_sum
                emit($k, [v_1, v_2]$)
                previous_sum = current_sum
            end
        end
    end

Procedure Reduce($k, [v_1, v_2]$)
    if $\sum v_2 \leq B_k$ **then**
        return 0
    else
        return minimal threshold v such that $\sum_{v_1 \geq v} v_2 \leq B_k$
    end

initialize λ^0
for $t \leftarrow 0$ **to** $T - 1$ **do**
    for each $i \in \{1, \ldots, N\}$ in parallel **do**
        Map(p_i, b_i, λ^t)
    end
    for each $k \in \{1, \ldots, K\}$ in parallel **do**
        $\lambda_k^{t+1} = $ Reduce($k, [v_1, v_2]$)
    end
    return λ^t if λ^t is converged
end

Algorithm 4: Synchronous coordinate descent for solving KPs.

4.4 Convergence

For the special case of $K = 1$, it is straightforward to show that our Algorithm 4 converges to a solution whose total profit is at most $max_{i,j} p_{i,j}$ less than the optimal solution (because it produces a rounded integer solution of a *fractional knapsack problem*) [6].

For more general cases, if the algorithm converges to a pair (x^*, λ^*) that jointly satisfy (5)–(10) (i.e., so-called "sufficient" optimality conditions), x^* is then optimal for the primal problem [19]. However, there is no theoretical guarantee on the convergence of

the algorithm to such a pair (x^*, λ^*). In fact, the solution x computed for the maximization problem (5)–(7) is not guaranteed to be even feasible for the problem (1)–(4). Nevertheless, as shown in [19], the solution x computed for (5)–(7) is optimal for a family of IP problems if B_k is replaced by $\sum_{i=1}^{N} \sum_{j=1}^{M} b_{i,j,k} x_{i,j} + \delta_k$ where δ_k is non-negative (equals to 0 when $\lambda_k > 0$). We analyze the optimality gap empirically and show our algorithms are often nearly optimal. In particular, the optimality gap decreases as the number of users N increases. When $K \ll N$ (which is often the case in real-world applications [9, 10, 21]), the optimality gap is so small that the resultant solution is very close to optimal.

5 FURTHER SPEEDUPS

5.1 Linear-time λ Candidate Generation

The calculation of candidate λ values for each group in the generalized algorithm has a time complexity of $O(KM^3 \log M)$. This can be further enhanced when global constraints are in *sparse* form and one local constraint exists per group such that:

- there exists a one-to-one mapping between the items and the knapsacks (i.e., $M = K$, and $b_{i,j,k} = 0, \forall j \neq k$), and
- there is one local constraint limiting the maximum number (denoted by Q thereafter) of items chosen for each group.

For such cases, there is *at most* one candidate for λ_k depending on whether the k-th item has a top Q adjusted profit or not. In particular, if the adjusted profit of item k is already in top-Q list, the critical value of new λ_k is the one that lowers its adjusted profit to the $(Q + 1)$-th adjusted profit. If the adjusted profit of item k is not in top Q, the critical value of new λ_k is the one that increases its adjusted profit to the Q-th adjusted profit. The pseudo-code is given in Algorithm 5 where $\bar{p}$ is the threshold deciding whether the k-th item will be chosen for group i. If new λ_k is larger than $\frac{p_{i,k}-\bar{p}}{b_{i,k,k}}$, the updated adjusted profit of the k-th item will be below $\bar{p}$ and thus the item will not be chosen. On the other hand, a new λ_k below $\frac{p_{i,k}-\bar{p}}{b_{i,k,k}}$ guarantees that the resulting adjusted profit of the k-th item is among top Q across all items. Thus, Algorithm 5 correctly emits the only candidate of new λ_k (if any) that determines whether the k-th item has a top Q adjusted profit or not.

Algorithm 5 uses $quick\_select(array, n)$ to find the n-th largest element of an K-array. The overall complexity of this procedure is $O(K)$, independent of the value of Q [4].

5.2 Fine-tuned Bucketing

A straightforward implementation of the *Reduce* function in Algorithm 4 is to sort the emitted results by the value of v_1 and choose the minimal threshold v based on the sorted results. A speedup is to bucket the values of v_1 and calculate the sum of v_2 for each bucket. We then identify the target bucket that the threshold v falls into, and approximate the value of v, for example, by interpolating within the bucket.

To improve the accuracy of the above approximation through bucketing and interpolating, we would like that the bucket size is small around the true value of v and large when the bucket is unlikely to contain v. Unfortunately, the true value of v is unknown to us beforehand. Nevertheless, due to the iterative nature of Algorithm 4, the value calculated in the previous iteration, i.e. λ_k^t,

---

Procedure Map(p_i, b_i, λ^t)

    initialize adjusted_profits as an array of K numbers

    foreach $k \in \{1, \ldots, K\}$ **do**

        adjusted_profits[k] = max($p_{i,k} - \lambda_k^t b_{i,k,k}, 0$)

    end

    Q_th_largest = quick_select(adjusted_profits, Q)

    Q1_th_largest = quick_select(adjusted_profits, $Q + 1$)

    foreach $k \in \{1, \ldots, K\}$ **do**

        if *adjusted_profits[k]* $\geq$ *Q_th_largest* **then**

            $\bar{p}$ = Q1_th_largest

        else

            $\bar{p}$ = Q_th_largest

        end

        if $p_{i,k} > \bar{p}$ **then**

            $v_1 = \frac{p_{i,k}-\bar{p}}{b_{i,k,k}}$

            $v_2 = b_{i,k,k}$

            emit($k, [v_1, v_2]$)

        end

    end

Algorithm 5: Linear time *Map* function for choosing up to Q items, each corresponding to a knapsack.

---

provides a reasonable estimate for $v = \lambda_k^{t+1}$. We thus designed an uneven bucketing scheme such that the bucket is of a minimal size around λ_k^t and grows exponentially as it deviates from λ_k^t. Specifically, given the value calculated in the previous iteration λ_k^t, the bucket id assigned to a candidate λ_k is given as

$$bucket\_id(\lambda_k) = sign(\lambda_k - \lambda_k^t) \lfloor \log \frac{|\lambda_k - \lambda_k^t|}{\Delta} \rfloor,$$

where Δ is a parameter controlling bucket sizes.

5.3 Pre-solving by Sampling

Like all other iterative algorithms, starting from a good initialization can significantly accelerate the convergence of the algorithm. The initial value for the dual multipliers, λ^0 in Algorithm 4, is often chosen randomly, but can be estimated by pre-solving using sampled data. By sampling small sets of random groups and adjusting knapsack budgets proportionally, we can start the algorithm with better initialization. Our experiments show that pre-solving can save up to 40% to 75% of iterations for large-scale KPs.

5.4 Post-processing for Feasibility

For a converged solution from our algorithms, it's likely that the total resource consumption violates the global constraints (2) just by a tiny bit. To ensure the satisfaction of the global constraints and also accelerate the convergence, we propose a light-weight post-processing method based on the *cost-adjusted group profit* quantity,

$$\tilde{p}_i = \sum_{j=1}^{M} p_{i,j} x_{i,j} - \sum_{k=1}^{K} \lambda_k \sum_{j=1}^{M} b_{i,j,k} x_{i,j}.$$

which is actually also the dual value contributed by a given group i. The post-processing procedure works by sorting the groups $\{i\}$ according to the non-decreasing order of $\tilde{p}_i$, and in this order one by

one, resetting x_i to 0 until all global constraints are satisfied. Since the cost-adjusted group profit measures the benefit of choosing items from group i, removing those with smaller values of $\tilde{p}_i$ until constraint satisfaction is a sensible heuristic to *project* the solution to its nearest neighbor (a boundary point) in the feasible region.

6 EXPERIMENTS

We tested the distributed solvers on both synthetic and real-world data[4]. Unless otherwise stated, for synthetic instances $p_{i,j}$ is uniformly distributed in [0, 1]. Two classes of global constraints (*sparse* and *dense*) are experimented with, where the cost coefficient $b_{i,j,k}$ is uniformly sampled within [0, 1]. Without losing generality, the budgets of global constraints are scaled with M, N and L to ensure tightness of constraints, and C_l) are set to 1.

In the following, *optimality ratio* is defined as the ratio of primal objective value to relaxed LP objective value; *constraint violation ratio* is defined as the ratio of excessive budget to given budget for a constraint, and we use *max constraint violation ratio* over all constraints to quantify overall violation of a solution.

6.1 Optimality Evaluation

We evaluate the optimality ratio between our KP solution against relaxed LP solution (i.e., the upper bound). It is difficult to find any existing LP solver that can handle billion-scale problems, therefore our comparison has to be restricted to modest-size problems so we can get the upper-bound using LP solvers such as Google OR-tools [18].

Figure 1 shows the optimality ratios for $N = 1,000$ and $10,000$ across $K \in \{1, 5, 10, 15, 20\}$. We fix the number of items per group M at 10. To increase the diversity of the items, $b_{i,j,k}$ are uniformly distributed in [0, 1] and in [0, 10] with equal probabilities (each being 0.5). We evaluated optimality ratios for three scenarios of local constraints as shown in Figure 1 where

- C=[1] means $\sum_{j=1}^{M} x_{i,j} \leq 1$,
- C=[2] means $\sum_{j=1}^{M} x_{i,j} \leq 2$, and
- C=[2,2,3] corresponds to hierarchical local constraints.

We plot the average optimality ratio (across 3 runs) as we vary K, as shown in Figure 1. The optimality gap decreases as N increases. The optimality ratio is above 98.6% for all experiment cases, and above 99.8% for $N = 10,000$ under all scenarios of local constraints.

6.2 Duality Gap on Large Datasets

We tested our system with large synthetic data to measure solution quality as well. The large-scale test sets contain a number of sparse global constraints with $N = 100$ million users, while the number of items M in each group varies from 1 to 100 and thus the total number of items we consider is up to 10 billion. Table 1 shows the number of SCD iterations, primal objective values and duality gaps[5][2]. The duality gaps are much smaller than the primal objective values, indicating that the solutions are nearly optimal. Furthermore, no global constraint is violated when our algorithm converges.

[4]In our typical real-world applications there are hundreds of millions users. As long as we are aware, no datasets of comparable size are publicly available.
[5]A duality gap is the difference between the dual objective value and the primal IP objective value.
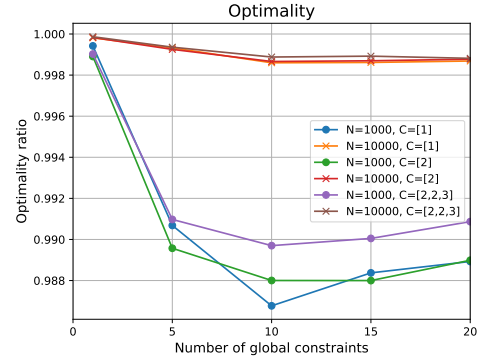

Figure 1: Optimality ratio between KP solutions and upper bounds computed by LP relaxation

Table 1: Results for 100 million users with up to 10 billion items

M	Iterations	Primal value	Duality gap
1	2	40,631,183.07	0.0
5	13	73,000,742.85	302.71
10	18	85,378,580.47	290.74
20	14	92,415,786.11	301.04
100	10	98,436,146.56	225.15

Table 2: Number of SCD iterations with/without pre-solving

| N	No pre-solving	Pre-solving	% of reduction
1 million	35	21	40%
10 million	32	8	75%
100 million	32	13	59%

6.3 Effectiveness of Pre-solving

When the number of groups N is large, pre-solving with sampled users is used to generate good starting points for λ. We sample $n = 10,000$ groups for pre-solving, and the computation time of pre-solving is negligible since $n \ll N$. We compare the number of SCD iterations until convergence with pre-solving and without pre-solving, both starting at $\lambda_k = 1.0, \forall k \in [K]$. Table 2 reports the number of SCD iterations for sparse problem instances with $N = 1$ million, 10 million, 100 million. For each N, we fix $M = 10$ and $K = 10$. The results in Table 2 show that pre-solving reduced the number of SCD iterations by 40% to 75%.

We also notice pre-solving alone is not sufficient for solving KP problems, as λ produced by pre-solving led to constraint violations. When applying pre-solved λ to full datasets, we observed that the number of global constraint violations are 4, 5 and 3 out of 10, for $N = 1$ million, 10 million, 100 million, respectively, and the corresponding max constraint violation ratio is 2.5%, 4.1% and 3.0%, respectively. Comparatively, the distributed SCD solutions have no violations. It is also worthwhile noting that the primal objective
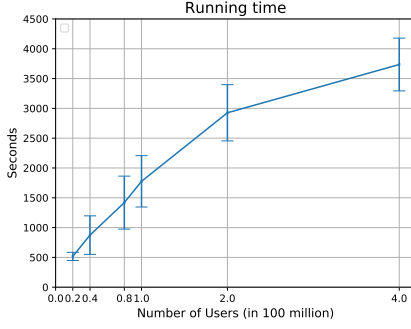
Figure 2: Running time with $N = 20, 40, 80, 100, 200, 400$ **million users on 10 dense global constraints and 200 executors**
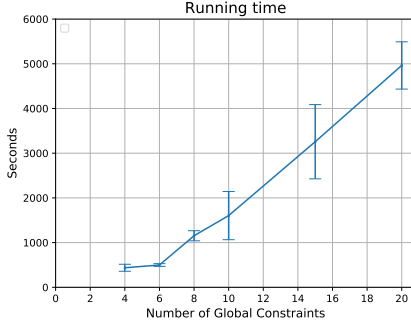
Figure 3: Running time with $K = 4, 6, 8, 10, 15, 20$ **global dense constraints on 100 million users**

Figure 4: Running time of speedup algorithm (speedup) and generalized algorithm (regular)
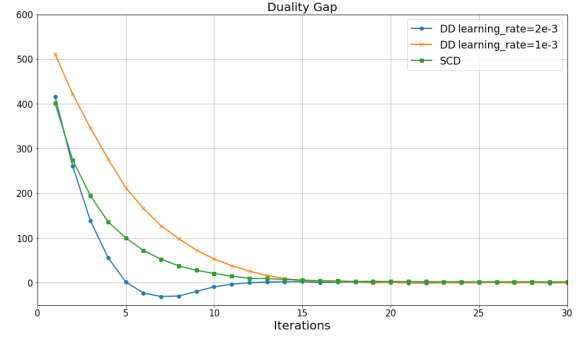
Figure 5: Duality gaps for DD and SCD

Figure 6: Max constraint violation ratios for DD and SCD

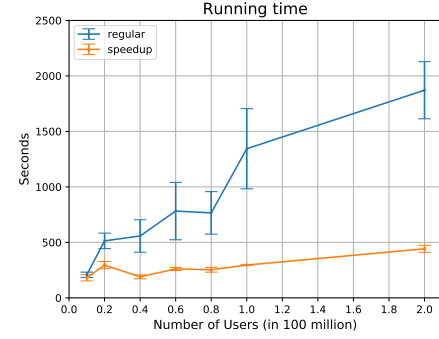of pre-solving solution, even with constraint violations, is always smaller than the SCD solution.

6.4 Scalability

To study the scalability of our solver, an implementation of our algorithm in Spark was tested using sparse and dense problem instances. Each Spark executor has 8 cores and 16G memory, and the number of executor is 200. Figure 2 shows running time with $N = 20, 40, 80, 100, 200, 400$ million users with $K = 10$ dense global constraints and hierarchical local constraints. Figure 3 plots running time with $K = 4, 6, 8, 10, 15, 20$ global constraints, with N fixed at 100 million. Figure 2 and 3 illustrate that Algorithm 4 is clearly scalable. Figure 4 demonstrates that speedup algorithm in Section ?? reduces the running time significantly and consistently across different number of users with 10 global constraints.

The running time of our system has satisfied the business need of daily optimization of the decision variables. For example, when running on Spark with 200 executors in a shared Apache Hadoop computing infrastructure, the optimization for 1 billion decision variables and constraints was able to converge within an hour.

6.5 Comparison of DD and SCD

To compare DD and SCD, both algorithms were run on sparse problem instances with $N = 10,000$, $M = 10$ and $K = 10$. We

experimented with a range of learning rates for DD, and present the results of learning rates being 1e-3 and 2e-3 here, as their convergence was most comparable to SCD in our experiment setting. Figure 5 plots the duality gaps as the number of iterations increases for the algorithms, while Figure 6 shows the corresponding max constraint violation ratios. The results show that the number of iterations taken by both algorithms are comparable, but for SCD the max constraint violation ratio is much smaller and also way more smooth. SCD also requires less parameter tuning effort across problem instances, and is thus used for our real-world jobs.

6.6 Production Deployment

Ant Financial provides its users a wide spectrum of financial products and services ranging from payment to banking, loans, wealth management, insurances and so on. As of late 2018, Ant is serving over 1 billion active users globally through its mobile payment Apps such as Alipay. Solving KPs at scale is crucial to our business as various of financial resources are being allocated among our users on a daily basis. The distributed algorithms we developed in this paper were deployed to production in late 2018 and ever since have bee used to power production decision making everyday for more than 10 of our core products, e.g., marketing budget allocation, insurance pricing, online traffic control, notification volume optimization, credit risk management, and loan allocation, etc.

7 CONCLUSIONS

We introduce distributed algorithms for solving billion-scale knapsack problems. Our approach is developed based on a slightly generalized formulation of KPs and hence can be applied to solve other variants of KPs. The proposed algorithms can be easily implemented using common distributed frameworks such as MPI, Hadoop and Spark. The approach can also be extended to solve KPs with nonlinear objective functions as long as it is decomposable with respect to the decision variables (or groups of decision variables).

REFERENCES

[1] Deepak Agarwal, Bee-Chung Chen, Pradheep Elango, and Xuanhui Wang. Personalized click shaping through lagrangian duality for online recommendation. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval, August 12 - 16, 2012*, pages 485 – 494, Portland, OR, USA, 2012.

[2] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004.

[3] P. C. Chu and John E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, USA, 3rd edition, 2009.

[5] CPlex. www.cplex.com.

[6] George B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–277, April 1957.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008.

[8] Fred Glover. Tabu search - part i. *ORSA Journal on Computing*, 1:135–206, 1989.

[9] Rupesh Gupta, Guanfeng Liang, and Rómer Rosales. Optimizing email volume for sitewide engagement. In *Proceedings of the 26th ACM CIKM International Conference on Information and Knowledge Management, November 6 - 10, 2017*, pages 1947 – 1955, Singapore, 2017.

[10] Rupesh Gupta, Guanfeng Liang, Hsiao-Ping Tseng, Ravi Kiran Holur Vijay, Xiaoyu Chen, and Rómer Rosales. Email volume optimization at linkedin. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 13 - 17, 2016*, pages 97 – 106, San Francisco, CA, USA, 2016.

[11] Gurobi. www.gurobi.com.

[12] Skander Htiouech and Sadok Bouamama. Osc: Solving the multidimensional multi-choice knapsack problem with tight strategic oscillation using surrogate constraints. *International Journal of Computer Applications*, 73:1–22, 2013.

[13] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, Berlin, 2004.

[14] Shahadatullah Khan. *Quality Adaptation in a Multisession Multimedia System: Model, Algorithms and Architecture*. PhD thesis, Department of Electrical and Computer Engineering, University of Victoria, Canada, 1998.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[16] Soukaina Laabadi, Mohamed Naimi, Hassan E. Amri, and Boujemâa Achchab. The 0/1 multidimensional knapsack problem and its variants: A survey of practical models and heuristic approaches. *American Journal of Operations Research*, 8:395–439, 2018.

[17] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, 1990.

[18] Laurent Perron and Vincent Furnon. Google or-tools, https://developers.google.com/optimization.

[19] Jeremy F. Shapiro. A survey of lagrangean techniques for discrete optimization. *Annals of Discrete Mathematics*, 5:113–138, 1979.

[20] Stephen J. Wright. Coordinate descent algorithms. *Mathematical Programming, Series B*, 151:3–34, 2015.

[21] Bo Zhao, Koichiro Narita, Burkay Orten, and John Egan. Notification volume control and optimization system at pinterest. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 19 - 23, 2018*, pages 1012 – 1020, London, United Kingdom, 2018.