Noname manuscript No.
(will be inserted by the editor)

Coupled and k-Sided Placements: Generalizing Generalized Assignment*

Madhukar Korupolu · Adam Meyerson · Rajmohan Rajaraman · Brian Tagiku

the date of receipt and acceptance should be inserted later

Abstract In modern data centers and cloud computing systems, jobs often require resources distributed across nodes providing a wide variety of services. Motivated by this, we study the *Coupled Placement* problem, in which we place jobs into computation and storage nodes with capacity constraints, so as to optimize some costs or profits associated with the placement. The coupled placement problem is a natural generalization of the widely-studied generalized assignment problem (GAP), which concerns the placement of jobs into single nodes providing one kind of service. We also study a further generalization, the *k-Sided Placement* problem, in which we place jobs into k-tuples of nodes, each node in a tuple offering one of k services.

For both the coupled and k-sided placement problems, we consider minimization and maximization versions. In the minimization versions (MinCP and Min$k$SP), the goal is to achieve minimum placement cost, while incurring a minimum blowup in the capacity of the individual nodes. Our first main result is an algorithm for Min$k$SP that achieves optimal cost while increasing capacities by at most a factor of $k + 1$, also yielding the first constant-factor approximation for MinCP. In the maximization versions (MaxCP and Max$k$SP), the goal is to maximize the total weight of the jobs that are placed under hard capacity constraints. Max$k$SP can be expressed as a k-column sparse integer program, and can be approximated to within a factor of $O(k)$

* A preliminary version of this paper appears in the *Proceedings of the 17th Conference on Integer Programming and Combinatorial Optimization*, 2014.

M. Korupolu
Google, 1600 Amphitheater Parkway, Mountain View, CA. E-mail: mkar@google.com

A. Meyerson
Google, 1600 Amphitheater Parkway, Mountain View, CA. E-mail: awmeyerson@google.com

R. Rajaraman
Northeastern University, Boston, MA 02115. E-mail: rraj@ccs.neu.edu

B. Tagiku
Google, 1600 Amphitheater Parkway, Mountain View, CA. E-mail: btagiku@google.com

factor using randomized rounding of a linear program relaxation. We consider alternative combinatorial algorithms that are much more efficient in practice. Our second main result is a local search based combinatorial algorithm that yields a 15-approximation and $O(k^2)$-approximation for MaxCP and Max$k$SP respectively. Finally, we consider an online version of Max$k$SP and present algorithms that achieve logarithmic competitive ratio under certain necessary technical assumptions.

1 Introduction

The data center has become one of the most important assets of a modern business. Whether it is a private data center for exclusive use or a shared public cloud data center, the size and scale of the data center continues to rise. As a company grows, so too must its data center to accommodate growing computational, storage and networking demand. However, the new components purchased for this expansion need not be the same as the components already in place. Over time, the data center becomes quite heterogeneous [23]. This complicates the problem of placing jobs within the data center so as to maximize performance.

Jobs often require resources of more than one type: for example, computation and storage. Modern data centers typically separate computation from storage and interconnect the two using a network of switches. As such, when placing a job within a data center, we must decide which computation node and which storage node will serve the job. If we pick nodes that are far apart, then communication latency may become too prohibitive. On the other hand, nodes are capacitated, so picking nodes close together may not always be possible.

Most prior work in data center resource management is focussed on placing one type of resource at a time: e.g., placing storage requirements assuming job computation location is fixed [14,3] or placing computation requirements assuming job storage location is fixed [4,11]. One sided placement methods cannot suitably take advantage of the proximities and heterogeneities that exist in modern data centers. For example, a database analytics application requiring high throughput between its computation and storage elements can benefit by being placed on a storage node that has a nearby available computation node.

In this paper, we study *Coupled Placement* (CP), which is the problem of placing jobs into computation and storage nodes with capacity constraints, so as to optimize costs or profits associated with the placement. Coupled placement was first addressed in [20] in a setting where we are required to place all jobs and we wish to minimize the communication latency over all jobs. They show that this problem, which we call MinCP, is NP-hard and investigate the performance of heuristic solutions. Another natural formulation is where the goal is to maximize the total number of jobs or revenue generated by the placement, subject to capacity constraints. We refer to this problem as MaxCP.

We also study a generalization of Coupled Placement, the *k-Sided Placement Problem* (kSP), which considers $k \geq 2$ kinds of resources.

1.1 Problem definition

In the *coupled placement* problem, we are given a bipartite graph $G = (U, V, E)$ where U is a set of compute nodes and V is a set of storage nodes. We have capacity functions $C : U \to \mathcal{R}$ and $S : V \to \mathcal{R}$ for the compute and storage nodes, respectively. We are also given a set T of jobs, each of which needs to be allocated to one compute node and one storage node. Each job may prefer some compute-storage node pairs more than others, and may also consume different resources at different nodes. To capture these heterogeneities, we have for each job j a function $f_j : E \to \mathcal{R}$, a computation requirement $p_j : E \to \mathcal{R}$ and a storage requirement $s_j : E \to \mathcal{R}$.

We consider two versions of the coupled placement problems. For the minimization version MinCP, we view f_j as a cost function. Our goal is to find an assignment $\sigma : T \to E$ such that all capacities are observed and our total cost $\sum_{j \in T} f_j(\sigma(j))$ is minimized. For the maximization version MaxCP, we view f_j as a profit function. Our goal is to select a subset $A \subseteq T$ of jobs and an assignment $\sigma : A \to E$ such that all capacities are observed and our total profit $\sum_{j \in A} f_j(\sigma(j))$ is maximized. In both problems, the constraint that "all capacities are observed" is formally captured by the requirement that $\sum_{j \in T : \sigma(j) \in \delta(u)} p_j(\sigma(j)) \leq C(u)$ for all $u \in U$ and $\sum_{j \in T : \sigma(j) \in \delta(v)} s_j(\sigma(j)) \leq S(v)$ for all $v \in V$, where $\delta(x)$ is the set of edges adjacent to node x in G.

A generalization of the coupled placement problem is *k-sided placement* (kSP), in which we have k different sets of nodes, $S_1, \ldots, S_k$, each set of nodes providing a distinct service. For each i, we have a capacity function $C_i : S_i \to \mathcal{R}$ that gives the capacity of a node in S_i to provide the ith service. We are given a set T of jobs, each of which needs each kind of service; the exact resource needs may depend on the particular k-tuple of nodes from $\prod_i S_i$ to which it is assigned. That is, for each job j, we have a demand function $d_j : \prod_i S_i \to \mathcal{R}^k$. We also have another function $f_j : \prod_i S_i \to \mathcal{R}$. As for coupled placement, we can assume that the capacities are unit, since we can scale the demands of individual nodes accordingly.

Similar to coupled placement, we consider two versions of kSP, Min$k$SP and Max$k$SP. In Min$k$SP, we view f_j as a cost function, and the goal is to find an assignment $\sigma : T \to \prod_i S_i$ such that all capacities are observed and the total cost $\sum_{j \in T} f_j(\sigma(j))$ is minimized. In Max$k$SP, we view f_j as a profit function, and the goal is to find an assignment $\sigma : T \to \prod_i S_i$ such that all capacities are observed and total profit $\sum_{j \in A} f_j(\sigma(j))$ is maximized. In both problems, "all capacities are observed" is formally captured by the requirement that for $1 \leq i \leq k$, for all $u \in S_i$, we have $\sum_{j \in T : u = \pi_i(\sigma(j))} \pi_i(d_j(\sigma(j))) \leq C_i(u)$, where for any k-dimensional vector $\mathbf{r}$, $\pi_i(\mathbf{r})$ is the value of the ith coordinate of $\mathbf{r}$.

1.2 Our Results

All of the variants of CP and kSP are NP-hard, so our focus is on approximation algorithms. Our first set of results consist of the first non-trivial approximation algorithms for MinCP and Min$k$SP. Under hard capacity constraints, a reduction from the NP-complete Partition problem shows that it is NP-hard to determine if there is a feasible solution to the problem; hence, it is NP-hard to achieve any bounded approximation ratio to cost minimization. So we consider approximation algorithms that incur a blowup in capacity. We say that an algorithm is α-approximate for the minimization version if its cost is at most that of an optimal solution, while incurring a blowup factor of at most α in the capacity of any node.

- We present a $(k+1)$-approximation algorithm for Min$k$SP using iterative rounding, yielding a 3-approximation for MinCP.

We note that in recent independent work, Harris and Srinivasan have developed a novel algorithmic framework for certain generalized scheduling problems, using the Lovász Local Lemma [16–18]. Their framework also applies to Min$k$SP, yielding an $O(\log k/\log\log k)$-approximation algorithm for the problem [7].

We next consider the maximization version. Max$k$SP can be expressed as a k-column sparse integer packing program (k-CSP), i.e., an integer program in which each column in the constraint matrix has at most k non-zero entries. From this, it is immediate that Max$k$SP can be approximated to within an $O(k)$ approximation factor by applying randomized rounding to a linear programming relaxation [8]. An $\Omega(k/\log k)$-inapproximability result for k-set packing due to [19] implies the same hardness result for Max$k$SP. Our second main result is a simpler approximation algorithm for MaxCP and Max$k$SP based on local search.

- We present local search based approximation algorithms for MaxCP and Max$k$SP, obtaining 15- and $O(k^2)$-approximations, respectively.

The local search result applies directly to a version where we can assign jobs fractionally but only to a single pair of machines. We then describe a simple rounding scheme to obtain an integral version. The rounding technique involves establishing a one-to-one correspondence between fractional assignments and machines. This is much like the cycle-removing rounding for GAP; there is a crucial difference, however, since coupled placements assign jobs to pairs of machines.

Finally, we study the online version of MaxCP and Max$k$SP, in which jobs arrive online and must be irrevocably assigned or rejected immediately upon arrival.

- We extend the techniques of [5] designed for competitive on-line routing and scheduling to our scenario where the resource requirement for a job can be arbitrarily machine-dependent. Under two necessary technical assumptions

– one about profit values of jobs, relative to their resource requirements, and the other about maximum resource requirements of a job – we design online algorithms that achieve optimal logarithmic competitive ratios for MaxCP and Max$k$SP.

1.3 Related Work

The coupled and k-sided placement problems are natural generalizations of the Generalized Assignment Problem (GAP), which can be viewed as a 1-sided placement problem. In GAP, which was first introduced by Shmoys and Tardos [24], the goal is to assign items of various sizes to bins of various capacities. A subset of items is feasible for a bin if their total size is no more than the bin's capacity. If we are required to assign all items and minimize our cost (MinGAP), Shmoys and Tardos [24] give an algorithm for computing an assignment that achieves optimal cost while doubling the capacities of each bin. Our $(k+1)$-approximation for Min$k$SP can be viewed as a generalization of the result for MinGAP, which corresponds to the case $k = 1$. A previous result by Lenstra *et al.* [22] for scheduling on unrelated machines show it is NP-hard to achieve optimal cost without incurring a capacity blowup of at least 3/2. On the other hand, if we wish to maximize our profit and are allowed to leave items unassigned (MaxGAP), Chekuri and Khanna [12] observe that the $(1, 2)$-approximation for MinGAP implies a 2-approximation for MaxGAP. This can be improved to a $(\frac{e}{e-1})$-approximation using LP-based techniques [15]. It is known that MaxGAP is APX-hard [12], though no specific constant of hardness is shown.

On the experimental side, most prior work in data center resource management focusses on placing one type of resource at a time: for example, placing storage requirements assuming job computation location is fixed (file allocation problem [14], [1–3]) or placing computation requirements assuming job storage location is fixed [4, 11]. These in a sense are variants of GAP. The only prior work on Coupled Placement is [20], where they show that MinCP is NP-hard and experimentally evaluate heuristics: in particular, a fast approach based on stable marriage and knapsacks is shown to do well in practice, close to the LP optimal.

The concept of k-sided placement problem also generalizes the concept of scheduling multidimensional jobs on unrelated machines: each dimension corresponds to a side, and for each machine in the multidimensional scheduling instance, we can associate a tuple of machines in the Min$k$SP instance, one machine from each side, and stipulate that each job can only be placed in one of these tuples. On the other hand, there is no straightforward reduction from k-sided placement to k-dimensional scheduling since the number of ways of placing job in k-sided placement can significantly exceed the number of machines. For the minimization version of k-dimensional scheduling problem, Azar and Epstein present a $(k + 1)$-approximation algorithm by essentially reducing the problem to a single-dimensional case [6].

The MAXkSP problem is related to the recently studied hypermatching assignment problem (HAP) [13], and special cases, including k-set packing, and a uniform version of the problem. A $(k + 1 + \varepsilon)$-approximation is given for HAP in [13], where other variants of HAP are also studied. While the MAXkSP problem can be viewed as a variant of HAP, there are critical differences. For instance, in MAXkSP, each job is assigned at most one tuple, while in the hypermatching problem each client (or job) is assigned a subset of the hyperedges. Hence, the MAXkSP and HAP problems are not directly comparable. The k-set packing can be captured as a special case of MAXkSP, and hence the $\Omega(k/\log k)$-hardness due to [19] applies to MAXkSP as well.

2 The minimization version

Next, we consider the minimization version of the Coupled Placement problem, MINCP. We write the following integer linear program for MINCP, where x_{tuv} is the indicator variable for the assignment of t to pair (u, v), $u \in U$, $v \in V$.

$$\begin{aligned}
\text{Minimize:} \quad & \sum_{t,u,v} x_{tuv} f_t(u,v) \\
\text{Subject to:} \quad & \sum_{u,v} x_{tuv} \geq 1, && \forall t \in T, \\
& \sum_{t,v} p_t(u,v) x_{tuv} \leq c_u, && \forall u \in U, \\
& \sum_{t,u} s_t(u,v) x_{tuv} \leq d_v, && \forall v \in V, \\
& x_{tuv} \in \{0,1\}, && \forall t \in T, u \in U, v \in V.
\end{aligned}$$

We refer the first set of constraints as *satisfaction* constraints, the second and third set as *capacity* constraints (computation and storage). We consider the linear relaxation of this program which replaces the integrality constraints above with $0 \leq x_{tuv} \leq 1, \forall t \in T, u \in U, v \in V$. Without loss of generality, we assume that $p_t(u, v) \leq c_u$ and $s_t(u, v) \leq d_v$ for all t, and (u, v); otherwise, we can set x_{tuv} to 0 and eliminate such triples from the linear program.

2.1 A 3-approximation algorithm for MINCP

We now present algorithm ITERROUND, based on iterative rounding [21], which achieves a 3-approximation for MINCP. We start with a basic algorithm that achieves a 5-approximation by identifying tight constraints with a small number of variables. Each iteration of this algorithm repeats the following round until all variables have been rounded.

1 **Extreme point:** Compute an extreme point solution x to the current LP.
2 **Eliminate variable or constraint:** Execute one of these two steps. By Lemma 3, one of these steps can always be executed if the LP is nonempty.

a Remove from the LP all variables x_{tuv} that take the value 0 or 1 in x. If x_{tuv} is 1, then assign job t to the pair (u, v), remove the job t and its associated variables from the LP, and reduce c_u by $p_t(u, v)$ and d_v by $s_t(u, v)$.
b Remove from the LP any tight capacity constraint with at most 4 variables.

Fix an iteration of the algorithm, and an extreme point x. Let n_t, n_c, and n_s denote the number of tight job satisfaction constraints, computation constraints, and storage constraints, respectively, in x. Note that every job satisfaction constraint can be assumed to be tight, without loss of generality. Let N denote the number of variables in the LP. Since x is an extreme point, if all variables in x take values in $(0, 1)$, then we have $N \leq n_t + n_c + n_s$.

Lemma 1 *If all variables in x take values in $(0, 1)$, then $n_t \leq N/2$.*

Proof Since a variable only occurs once over all satisfaction constraints, if $n_t > N/2$, there exists a satisfaction constraint that has exactly one variable. But then, this variable needs to take value 1, a contradiction.

Lemma 2 *If $n_t \leq N/2$, then there exists a tight capacity constraint that has at most 4 variables.*

Proof If $n_t \leq N/2$, then $n_s + n_c \geq N - n_t \geq N/2$. Since each variable occurs in at most one computation constraint and at most one storage constraint, the total number of variable occurrences over all tight storage and computation constraints is at most $2N$, which is at most $4(n_s + n_c)$. This implies that at least one of these tight capacity constraints has at most 4 variables.

Using Lemmas 1 and 2, we can argue that the above algorithm yields a 5-approximation. Step 2a does not cause any increase in cost or capacity. Step 2b removes a constraint, hence cannot increase cost; since the removed constraint has at most 4 variables, the total demand allocated on the relevant node is at most the demand of four jobs plus the capacity already used in earlier iterations. Since each job demand is at most the capacity of the node, we obtain a 5-approximation with respect to capacity.

Studying the proof of Lemma 2 more closely, one can separate the case $n_t < N/2$ from the $n_t = N/2$; in the former case, one can, in fact, show that there exists a tight capacity constraint with at most 3 variables. Together with a careful consideration of the $n_t = N/2$ case, one can improve the approximation factor to 4. We now present an alternative selection of tight capacity constraint that leads to a 3-approximation. One interesting aspect of this step is that the constraint being selected may not have a small number of variables. We replace step 2b by the following.

2b Remove from the LP any tight capacity constraint in which the number of variables is at most two more than the sum of the values of the variables.

Lemma 3 *If all variables in x take values in $(0, 1)$, then there exists a tight capacity constraint in which the number of variables is at most two more than the sum of the values of the variables.*

Proof Since each variable occurs in at most two tight capacity constraints, the total number of occurrences of all variables across the tight capacity constraints is $2N - s$ for some nonnegative integer s. That is, s equals $2N$ minus the total number of occurrences of all the variables in the tight capacity constraints. If we let v_0, v_1, and v_2 denote the number of variables that appear zero times, once, and twice, respectively, in the tight capacity constraints, then N equals $v_0 + v_1 + v_2$ while s equals $v_1 + 2v_0$.

Adding over all the tight satisfiability constraints, and doubling each side yields twice the sum of the values of the distinct variables on one side and $2n_t$ on the other. Since v_1 variables appear once and v_0 variables do not appear in the tight capacity constraints, and each variable takes on value less than 1, twice the sum of the values of the distinct variables is at most the sum of all variables, with multiplicity, in tight capacity constraints and s. We thus obtain that the sum of all variable values in tight capacity constraints is at least $2n_t - s$.

Therefore, the sum, over all tight capacity constraints, of the difference between the number of variables and their sum is at most $2(N - n_t)$. Since there are $N - n_t$ tight capacity constraints, for at least one of these constraints, the difference between the number of variables and their sum is at most 2.

Lemma 4 *Let u be a node with a tight capacity constraint, in which the number of variables is at most 2 more than the sum of the variables. Then, the sum of the capacity requirements of the jobs partially assigned to u is a most the current available capacity of u plus twice the capacity of u.*

Proof Let ℓ be the number of variables in the constraint for u, and let the associated jobs be numbered 1 through ℓ. Let the demand of job j for the capacity of node u be d_j. Then, the capacity constraint for u is $\sum_j d_j x_j = \widehat{c}(u)$, where $\widehat{c}(u)$ is the available capacity of u in the current LP.

We know that $\ell - \sum_i x_i \leq 2$. Since $d_i \leq C(u)$, the capacity of u:

$$\sum_j d_j = \widehat{c}(u) + \sum_{j=1}^{\ell}(1 - x_j)d_j \leq \widehat{c}(u) + (\ell - \sum_{j=1}^{\ell} x_j)C(u) \leq \widehat{c}(u) + 2C(u).$$

Theorem 1 IterRound *is a polynomial-time 3-approximation algorithm for* MinCP.

Proof By Lemma 3, each iteration of the algorithm removes either a variable or a constraint from the LP. Hence the algorithm is polynomial time. The elimination of a variable that takes value 0 or 1 does not change the cost. The elimination of a constraint can only decrease cost, so the final solution has cost no more than the value achieved by the original LP. Finally, when a capacity constraint is eliminated, by Lemma 4, we incur a blowup of at most 3 in capacity.

We note that our proof actually establishes a tighter upper bound on the capacity blowup, in terms of the maximum size of a job. In particular, our algorithm needs the capacity at any node to be at most its original capacity plus twice the largest requirement of a job that can be allocated to the node.

2.2 A $(k+1)$-approximation algorithm for MINkSP

It is straightforward to generalize the algorithm of the preceding section to obtain a $k+1$-approximation to MINkSP. We first set up the integer LP for MINkSP. For a given element $e \in \prod_i S_i$, we use e_i to denote the ith coordinate of e. Let x_{te} be the indicator variable that t is assigned to $e \in \prod_i S_i$.

$$\begin{aligned}
\text{Minimize:} \quad & \sum_{t,e} x_{te} f_t(e) \\
\text{Subject to:} \quad & \sum_{e} x_{te} \geq 1, && \forall t \in T, \\
& \sum_{t,e:e_i=u} (d_t(e))_i x_{te} \leq C_i(u), && \forall 1 \leq i \leq k, u \in U, \\
& x_{te} \in \{0,1\}, && \forall t \in T, e \in E
\end{aligned}$$

The algorithm, which we call ITERROUND(k), is identical to ITERROUND of Section 2.1 except that step 2b is replaced by the following.

2b Remove from the LP any tight capacity constraint in which the number of variables is at most k more than the sum of the values of the variables.

We now analyze ITERROUND(k). Fix an iteration of the algorithm, and an extreme point x. Let n_t denote the number of tight satisfaction constraints, and n_i denote the number of tight capacity constraints on the ith side. Since x is an extreme point, if all variables in x take values in $(0,1)$, then we have $N = n_t + \sum_i n_i$.

Lemma 5 *If all variables in x take values in $(0,1)$, then there exists a tight capacity constraint in which the number of variables is at most k more than the sum of the variables.*

Proof Since each variable occurs in at most k tight capacity constraints, the total number of occurrences of all variables across the tight capacity constraints is $kN - s$ for some nonnegative integer s. If we let v_i denote the number of variables that appear i times, $0 \leq i \leq k$, in the tight capacity constraints, then N equals $\sum_{i \geq 0} v_i$ while s equals $\sum_{i<k} (k-i)v_i$.

Adding over all the tight satisfiability constraints, and multiplying each side by k yields k times the sum of the values of the distinct variables on one side and kn_t on the other. Since v_i variables appear i times in the tight capacity constraints, and each variable takes on value less than 1, k times the sum of the values of the distinct variables is at most the sum of all variables, with multiplicity, in tight capacity constraints and s. We thus obtain that the sum of all variable values in tight capacity constraints is at least $kn_t - s$.

Therefore, the sum, over all tight capacity constraints, of the difference between the number of variables and their sum is at most $k(N - n_t)$. Since the number of tight capacity constraints is $N - n_t$, for at least one of these constraints, the difference between the number of variables and their sum is at most k.

Lemma 6 *Let u be a side-i node with a tight capacity constraint, in which the number of variables is at most k more than the sum of the variables. Then, the sum of the capacity requirements of the jobs partially assigned to u is at most the available capacity of u plus $kC_i(u)$.*

Proof Let ℓ be the number of variables in the constraint for u, and let the associated jobs be numbered 1 through ℓ. Let the demand of job j for the capacity of node u be d_j. Then, the capacity constraint for u is $\sum_j d_j x_j = \widehat{c}(u)$.

We know that $m - \sum_i x_i \leq k$. We also have $d_i \leq C_i(u)$. Letting $\widehat{c}(u)$ denote the current capacity of u, we now derive

$$\begin{aligned}\sum_i d_i &= \widehat{c}(u) + \sum_{j=1}^{m}(1 - x_i)d_i \\ &\leq \widehat{c}(u) + (m - \sum_{j=1}^{m} x_i)C_i(u) \\ &\leq \widehat{c}(u) + kC_i(u).\end{aligned}$$

Theorem 2 IterRound*(k) is a polynomial-time $k+1$-approximation algorithm for* Min*k*SP*.*

Proof By Lemma 5, each iteration of the algorithm removes either a variable or a constraint from the LP. Hence the algorithm is polynomial time. The elimination of a variable that takes value 0 or 1 neither changes cost nor incurs capacity blowup. The elimination of a constraint can only decrease cost, so the final solution has cost no more than the value achieved by the original LP. Finally, by Lemma 6, we incur a blowup of at most $1 + k$ in capacity.

As we observed for MinCP, our proof actually establishes a tighter upper bound on the capacity blowup, in terms of the maximum size of a job. In particular, our algorithm needs the capacity at any node to be at most its original capacity plus k times the largest requirement of a job that can be allocated to the node.

Integrality gap. A natural question to ask is whether a linear approximation factor for Min$k$SP is unavoidable for polynomial time algorithms. We now show that the Min$k$SP linear program has an integrality gap that grows as $\Omega(\log k / \log\log k)$. Determining the best efficiently achievable approximation factor for Min$k$SP is an open problem.

We recursively construct an integrality gap instance $\Gamma_{\ell,t}$ with ℓ^t sides, for parameters ℓ and t, with two nodes per side one with infinite capacity and the other with unit capacity, and t jobs, such that any integral solution has t jobs on the unit-capacity node on some side, while there is a fractional solution with load of at most t/ℓ on the unit-capacity node of each side. Setting $t = \ell$ and $k = \ell^\ell$, we obtain an instance in which the capacity used by the fractional solution is 1, while any integral solution has load $\ell = \Theta(\log k / \log\log k)$. Our construction is illustrated in Figure 1.

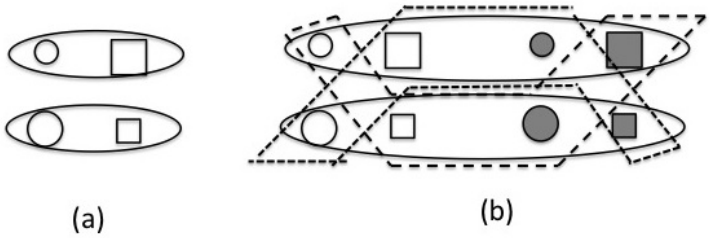

Fig. 1 (a) The construction $\Gamma_{\ell,t}$ for $t = 1$ and $\ell = 2$. The first side consists of two circle nodes, the small circle with unit capacity and the large circle with infinite capacity. The second side consists of two square nodes, the small square with unit capacity and the large square with infinite capacity. Job 1 can be placed on either the tuple consisting of the small circle and the large square or the large circle and the small square. (b) The construction $\Gamma_{\ell,t}$ for $t = 2$ and $\ell = 2$. One copy of the construction with 2 sides is unshaded, and the second copy is shaded. Small circles and squares have unit capacity, and large circles or squares have infinite capacity. There are two jobs. The first job can be placed according to one of two tuples, as indicated by the two ovals. The second job can be placed according to one of two tuples, as indicated by the two polygons. Any integral solution will incur a capacity blowup of at least two on some unit capacity node, while there exists a fractional solution (each job is placed half on each of its two tuples) that satisfies all capacity constraints.

Each job can be placed on one tuple from a subset of tuples; for a given tuple, the demand of the job on each side of the tuple is one. As induction base, we start with the construction for $t = 1$. We introduce a job that has ℓ choices, the ith choice consisting of the unit-capacity node from side i and infinite capacity nodes on all other sides. Clearly, in any integral solution the entire capacity of one unit-capacity node will be used, while there is a fractional solution ($1/\ell$ for each choice) in which only $1/\ell$ fraction of each unit capacity node will be used.

Given a construction $\Gamma_{\ell,t}$ with ℓ^t sides satisfying the desired properties, we show how to extend to a construction $\Gamma_{\ell,t+1}$ with ℓ^{t+1} sides using ℓ identical copies of $\Gamma_{\ell,t}$. In $\Gamma_{\ell,t+1}$, let us suppose without loss of generality that side $i\ell^t + r$ with $0 \leq i < \ell$ and $1 \leq r \leq \ell^t$ refers to side r of copy $i + 1$ (where the copies are numbered 1 through ℓ). The construction $\Gamma_{\ell,t}$ has t associated jobs. The construction $\Gamma_{\ell,t+1}$ has the t jobs from $\Gamma_{\ell,t}$ and an additional job that we introduce below. Informally, we combine the tuples for each of the t jobs in the different copies of $\Gamma_{\ell,t}$ in such a way that for any integral placement, for any $1 \leq r \leq \ell^t$, the nodes in side r (of each copy) on which j is placed are identical copies of one another. Formally, consider a job j in $\Gamma_{\ell,t}$. Suppose j can be placed in an ℓ^t-dimensional tuple $\mathbf{v}$ in which the entry for side r is node $\mathbf{v}_r$. Then, in $\Gamma_{\ell,t+1}$, j can be placed in an ℓ^{t+1}-dimensional tuple $\mathbf{w}$ where for $0 \leq i < \ell$ and $1 \leq r \leq \ell^t$, $\mathbf{w}_{i\ell^t+r}$ equals the node v_r in the copy $i + 1$ of Γ_t.

Finally, we add job $t + 1$ which can be placed in one of ℓ tuples: unit capacity node on all sides of copy i and infinite capacity node on all other sides, for each $1 \leq i \leq \ell$. We are ready to establish the integrality gap. By our

construction of $\Gamma_{\ell,t}$ (our induction hypothesis), any integral placement of the t jobs places a load of t on a unit-capacity node of some side, in each copy of $\Gamma_{\ell,t}$. Irrespective of which tuple we assign job $t+1$ to, any integral solution will have to add job $t+1$ to a unit-capacity node of a side that already has load t, yielding a load of $t+1$ on some node. On the other hand, a fractional solution assigns each job fractionally to the extent of $1/\ell$ to each of the ℓ tuples. This incurs a load of at most t/ℓ to the unit-capacity nodes of each side, yielding the desired integrality gap.

3 The maximization problems

We present approximation algorithms for the maximization versions of coupled placement and k-sided placement problems. We first observe, in Section 3.1, that these problems reduce to column sparse integer packing. We next present, in Section 3.2, an alternative combinatorial approach based on local search.

3.1 An LP-based approximation algorithm

One can write a positive integer linear program for MaxCP. Let x_{tuv} be the indicator variable for assigning job t to (u, v), $u \in U$, $v \in V$.

$$\begin{aligned}
\text{Maximize:} \quad & \sum_{t,u,v} x_{tuv} f_t(u,v) \\
\text{Subject to:} \quad & \sum_{u,v} x_{tuv} \leq 1, && \forall t \in T, \\
& \sum_{t,v} p_t(u,v) x_{tuv} \leq c_u, && \forall u \in U, \\
& \sum_{t,u} s_t(u,v) x_{tuv} \leq d_v, && \forall v \in V, \\
& x_{tuv} \in \{0,1\}, && \forall t \in T, u \in U, v \in V.
\end{aligned}$$

The above LP can be easily extended to Max$k$SP.

$$\begin{aligned}
\text{Maximize:} \quad & \sum_{t,e} x_{te} f_t(e) \\
\text{Subject to:} \quad & \sum_{e} x_{te} \leq 1, && \forall t \in T, \\
& \sum_{t,e} (d_t(e))_i x_{te} \leq C_i(e_i), && \forall i \in \{1, \ldots, k\}, \\
& x_{te} \in \{0,1\}, && \forall t \in T, e \in \textstyle\prod_i S_i.
\end{aligned}$$

These linear programs are 3- and k-column sparse packing programs, respectively; that is, each column in the constraint matrices of the linear programs has at most 3 and k non-zero entries, respectively. The 3- and k-column sparse packing integer linear programs can be approximated to within a factor

of 15.74 and $ek + o(k)$, respectively using a clever randomized rounding approach [8]. As mentioned in Section 1, an $\Omega(k/\log k)$-inapproximability result is known for MAXkSP.

3.2 Approximation algorithms based on local search

We now present a combinatorial approach for MAXCP based on local search, which is likely to be much more efficient than the above LP-based approximation algorithm in practice. Our approach also extends to MAXkSP, but we omit the details. The local search is somewhat unusual in that it is based on a *fractional* solution. Let x_{tuv} represent the fractional assignment of task t to nodes (u, v). The local search will maintain all linear program constraints, and will also maintain that $x_{tuv} > 0$ for at most one (u, v). Note that this last constraint is much more restrictive than the linear program.

A local step moves a single task t to (u, v). This can be thought of as first setting all the variables for t to zero (removing it from its current assignment, if any), and then gradually increasing x_{tuv} while possibly decreasing the fractional values for other tasks to insure that the capacity constraints at u and v are not violated. Supposing that increasing x_{tuv} would violate the constraint at u, we will decrease the value of $x_{t'uv'}$ such that the density value $f_{t'}(u, v)/p_{t'}(u, v)$ is minimimized (lowest value per unit capacity task t'). We continue until we have $x_{tuv} = 1$, or until increasing x_{tuv} further would actually reduce the objective (because of the need to decrease other tasks t' to maintain the constraints).

The local search algorithm starts with all $x_{tuv} = 0$ and proceeds by determining whether there exists any local step which would improve the objective value by at least $\epsilon\mu$,where μ is defined as $\mu = \frac{1}{4n} \max_{t,u,v} f_t(u, v)$. It continues making such local steps until none exists.

Lemma 7 *The local search algorithm satisfies all linear program constraints, and additionally guarantees that for each t, $x_{tuv} > 0$ for at most one pair (u, v).*

Proof The local step first dropped all variables for t to zero, which cannot violate any linear program constraint. Then while increasing x_{tuv} we were careful to always decrease some other $x_{t'uv'}$ if the constraint for u would be violated (and similarly to decrease some other $x_{t'u'v}$ if the constraint for v would be violated). It follows that no constraints are violated. The statement that for each t, $x_{tuv} > 0$ for at most one (u, v) is clear from the local step definition.

Theorem 3 *The local search algorithm produces a $3+\epsilon$ approximate fractional solution.*

Proof When the algorithm terminates, consider a task t which has $x_{tuv} = 1$ in the optimum solution. Consider the local step which would move t to

(u, v); how much would this increase the objective? We would lose the current objective value for t and obtain instead $f_t(u, v)$. However, we may also need to remove some other tasks from one or both of u and v in order to prevent violating the constraints. Let $F_u = \Sigma_t \Sigma_v f_t(u, v) x_{tuv}$ be the total value of tasks assigned to u. Then F_u/c_u is the average density of tasks assigned to u. Since we will always reduce the fraction of the lowest-density task assigned to u (when we must remove a task in order to prevent violating the constraints), the total value of tasks we need to remove from u in order to get t to fit cannot exceed $p_t(u, v)F_u/c_u$. Similarly the total value fo tasks we need to remove from v in order to get t to fit cannot exceed $s_t(u, v)F_v/d_v$. The total increase in objective is at least:

$$\Delta_t \geq f_t(u, v) - p_t(u, v)\frac{F_u}{c_u} - s_t(u, v)\frac{F_v}{d_v} - \Sigma_{u',v'} f_t(u', v') x_{tu'v'}$$

Note that while it's possible the algorithm would actually halt early when moving t to (u, v), it would only do so if the objective is decreasing, so Δ_t would be even large than the expression above (which was assuming we fully set $x_{tuv} = 1$. Since the local search algorithm terminated, we must have $\Delta_t < \epsilon\mu$. We combine inequalities and rearrange so everything is positive:

$$f_t(u, v) > p_t(u, v)\frac{F_u}{c_u} + s_t(u, v)\frac{F_v}{d_v} + \Sigma_{u',v'} f_t(u', v') x_{tu'v'} + \epsilon\mu$$

We now sum over all t, u, v with $x_{tuv} = 1$ in the optimum solution. The lefthand side will sum to the objective value of optimum. We observe that $\Sigma_u F_u$ and $\Sigma_v F_v$ are each the total objective found by the algorithm (measured by summing on one type of node only). Combined with the definition of μ and of the objective function, this gives us the approximation bound.

We remark that we can extend the above local search procedure to MAXkSP if we increase the approximation factor to $k + 1 + \epsilon$.

Theorem 4 *The local search algorithm runs in polynomial time.*

Proof Each local step increases the objective by at least $\epsilon\mu$. The optimum solution is at most $4n^2\mu$ (from the definition of μ, any single task attains value at most $4n\mu$ to the objective), so the algorithm terminates in at most $4n^2/\epsilon$ local steps. It's straightforward that a single local step can be executed in polynomial time (at most polynomially many tuples (t, u, v) to consider, at most polynomial time for each local step).

Rounding Phase: When the local search algorithm terminates, we have a fractional solution for MAXCP where each task t has at most one non-zero x_{tuv}. The next phase of the algorithm is to round the fractional solution returned by local search. Applying the randomized rounding approach of [8], we obtain an $O(k^2)$-approximation for MAXkSP, and a $(47.22+\varepsilon)$-approximation

for MaxCP. The preceding approach does not take advantage, however, of the properties of the fractional solution returned by our local search algorithm. For MaxCP, we present a different rounding scheme that exploits the local search invariants satisfied by the fractional solution and obtains a $15 + \epsilon$ approximation.

The idea behind the rounding scheme is to label each task which has a fractional assignment $0 < x_{tuv} < 1$ with one of the nodes u, v to which it is assigned in such a way that each label is allocated to at most one task. However, this requires a post-processing step after the local-search algorithm completes. Once this is done, we construct three integer solutions and argue that one of them must be $15 + \epsilon$ approximate. We will first prove a quick lemma about the structure of a graph based on node degrees.

Lemma 8 *Given a graph where every node has degree at least two and some node has degree at least three, we can find two cycles C_1 and C_2 in the graph. These cycles need not be disjoint, but they are distinct (so at least one edge must be in $C_1 - C_2$).*

Proof Consider only a single connected component of the graph, which includes at least one node with degree more than two. Start at this node, and traverse edges in sequence until we cannot do so without traversing an edge for the second time. This must halt since there are a finite number of edges. Since all nodes have degree at least two, at least one node in this traversal must be encountered twice. Select a part of this traversal where only one node is encountered twice; this gives us a simple cycle. Now consider deleting the edges of this cycle. Since the cycle includes at most two edges adjoining any node and there is a node of degree more than two, we have not removed all edges of the original connected component. If the remaining edges of the original component include a cycle, these two cycles can be C_1 and C_2 (and they are in fact edge-disjoint). If the remaining graph is acyclic, it must include a node of degree one (else we can find a cycle by the same traversal described above). Consider a traversal starting at such a node, where we again traverse the remaining edges one at a time until we can no longer do so. Since the graph is acyclic, this must end at another node of degree one. So we have a cycle C along with a path P which is edge-disjoint to C, with the endpoints of the path having degree one when we delete the nodes of C. Since all nodes originally had degree at least two, it follows that the endpoints of P must be nodes of C, say u, v. Cycle C includes a path from u to v (call it C_{uv}) and an edge-disjoint path from v back to u (call it C_{vu}). We can then set $C_1 = C_{uv} + P$ and $C_2 = C_{vu} + P$, both of which are cycles since P also connects u, v and is edge-disjoint from C. Since C_{uv}, C_{vu}, P are each edge-disjoint and non-empty, we must have at least on edge in $C_{uv} = C_1 - C_2$.

We are now ready to prove the main lemma in the analysis of our local search algorithm.

Lemma 9 *Given any fractional solution where each t has at most one non-zero assignment, we can produce an alternative fractional solution (with the*

same property and only larger objective) along with a labeling of each job t with $0 < x_{tuv} < 1$, with either u or v, guaranteeing that each node is the label for at most one job.

Proof We can remove the variables x_{tuv} which are zero or one in the local search solution. The remaining linear program has only the constraints at u and v (along with all variables between 0 and 1). It follows that there must exist a basic solution to this modified linear program with at least the objective obtained in the local search. This solution has the property described (via a simple flow argument between the variables and constraints). The remainder of this section discusses a combinatorial way to find a basic solution in the case where all variables appear in at most two equations (this will be the case here for $k = 2$).

Consider a multi-graph where the nodes are the storage and computation nodes. The edges (u, v) correspond to tasks where $0 < x_{tuv} < 1$. If any node (whether storage or computation) v has degree zero or one, we remove it and any adjoining edge, relabeling the edge with v. We continue this process until all remaining nodes have degree at least two. If all nodes have degree exactly two, then the graph is a simple cycle, and we label each edge with one of its endpoints (clockwise around the cycle).

Suppose there is a node of degree at least three. Then we can apply Lemma 8 to find two cycles C_1 and C_2 from the same connected component. Since the cycles are connected, we can find the shortest path P (fewest edges) between any node in C_1 and node in C_2 (this will be zero edges if the cycles are not node disjoint); let P connect $u_1 \in C_1$ to $u_2 \in C_2$ (it doesn't really matter if the nodes here are storage or computation nodes; the rest of the proof is identical).

We now show how to adjust the fractional assignments corresponding to the edges in the two cycles and the connecting path such that all capacity constraints are maintained and at least one more assignment becomes integral. We traverse the edges of C_1 one at a time, starting at node u_1. For the first edge, (u, v) we consider increasing x_{tuv} by ϵ. For the next edge (w, v), we select an ϵ' such that the total capacity used in the common node v is unchanged. Continuing around C_1, only the capacity used at the starting node u_1 will change. Suppose we use δ_1 additional capacity. If $\delta_1 = 0$ we can scale up all the ϵ values until some edge becomes integral. We perform the same operation on C_2, such that only the capacity used at the starting node u_2 will change, here by some value δ_2. The only case where we have not been able to make another assignment integral is when both δ_1 and δ_2 are non-zero. Now we traverse the edges of P, starting from u_1 and modifying the value of x on the first edge so that u_1 capacity will decrease by δ_1. We continue until we reach u_2, with capacity increasing by δ_2'. We now combine the adjustment to C_1 with the adjustment to P, and scale the adjustment to C_2 by $-\frac{\delta_2'}{\delta_2}$. The overall effect is that all capacity constraints are maintained, and we can scale the changes on the edges until some task becomes integrally assigned.

The above process decreases the number of fractional variables by one whenever there exists a node with degree greater than two. Combined with the labeling approach described when all nodes have degree at most two, we can produce the labeling described in the lemma.

Theorem 5 *There exists a polynomial-time algorithm based on local search that achieves a* $15 + \epsilon$ *approximation for* MaxCP.

Proof Given the labeled fractional solution of 9, we consider three possible solutions. Let A_I consist of all the integral assignments, A_U consist of the fractional assignments labeled with nodes of U, and A_V consist of the fractional assignments labeled with nodes of V. Clearly $A_I + A_U + A_V \geq \frac{1}{3}OPT$ is the value of the algorithm's solution. Of course, A_I is integral, but A_U, A_V will not be.

We can construct a solution S_U as follows. We consider the fractional solution corresponding to A_U. These are tuples (t, u, v) which are labeled with their node $u \in U$. Since at most one task has any label, each $u \in U$ appears at most once here; graphically this looks like a set of disjoint stars centered at the nodes $v \in V$. We select a suitable subset of tasks from the star centered at v such that we obtain at least half the value of the fractional solution for this star. This can always be done by selecting either the maximum density $(f_t(u, v)/s_t(u, v))$ tasks until we would overflow capacity d_v, or by selecting the maximum value single task. We repeat this for each v to obtain an integral solution S_U with value at least $A_U/2$. Similarly we can obtain solution S_V with value at least $A_V/2$. Thus we can get any of $A_I, A_U/2, A_V/2$ whichever has highest value. One of these must exceed $OPT/15$.

4 Online MaxCP and Max$k$SP

We now study the online version of MaxCP, in which jobs arrive in an online fashion. When a job arrives we must irrevocably assign it or reject it. Our goal is to maximize our total profit at the end of the instance. We apply the techniques of [5] to obtain a logarithmic competitive online algorithm under certain assumptions. The online MaxCP differs from the model considered in [5] in that a job's computation/storage requirements need not be the same, and may vary from one node to another. We note that the algorithm of [5] as well as our adaptation can also be cast within the elegant online primal-dual framework pioneered by Buchbinder and Naor [9,10].

As demonstrated in [5] certain assumptions have to be made to achieve competitive ratios of any interest. We extend these assumptions for the MaxCP model as follows:

Assumption 1 (Proportional profit) *There exists* F *such that for all* t, u, v *either* $f_t(u, v) = 0$ *or* $1 \leq f_t(u, v) \leq F \min(\frac{p_t(u,v)}{c_u}, \frac{s_t(u,v)}{d_v})$.

Assumption 2 (Small size) *For* $\epsilon = \min(\frac{1}{2}, \frac{1}{\ln(2F+1)+1})$, *for all* t, u, v: $p_t(u, v) \leq \epsilon c_u$ *and* $s_t(u, v) \leq \epsilon d_v$.

It is not hard to show that they (or some similar flavor of these assumptions) are in fact necessary to obtain any interesting competitive ratios.

Theorem 6 *No deterministic online algorithm for* MaxCP *or* Max$k$SP *can be competitive over classes of instances where either one of the following is true: (i) job size is allowed to be arbitrarily large relative to capacities, or (ii) job profits and resource requirements are completely uncorrelated.*

Proof We first show that if resource requirements are large compared to capacities, profit functions f_t are exactly equal to the total amount of resources and each job requires the same amount over all resources (but different jobs can require different amounts), then no deterministic online algorithm can be competitive.

Consider a graph G with a single compute node and a single data storage node. Each node has compute/storage capacity of L. A job arrives requesting 1 unit of computing and storage and will pay 2. Clearly, any competitive deterministic algorithm must accept this job, in case this is the only job. However, a second job arrives requesting L units of computing and storage and will pay $2L$. In this case, the algorithm is L-competitive, and L can be arbitrarily large.

Next, we show that if resource requirements are small relative to capacities, profit functions f_t are arbitrary and resource requirements are identical, then no deterministic online algorithm can be competitive. This instance satisfies Assumption 2 but not Assumption 1.

Consider again a graph G with a single compute node and single data storage node, each with unit capacity. We will use up to $k+1$ jobs, each requiring $1/k$ units of computing and storage. The i-th job, $0 \leq i \leq k$, will pay M^i for some large profit M. Now, consider any deterministic algorithm. If it fails to accept any job $j < k$, then if job j is the last job, it will be $\Omega(M)$-competitive. If the algorithm accepts jobs 0 up through $k-1$ then it will not be able to accept job k and will be $\Omega(M)$-competitive. In all cases it has competitive ratio $\Omega(M)$ and M and k can be arbitrarily large.

Similarly, if resource requirements are small relative to capacities, profit functions f_t are exactly equal to the total amount of resources requested and resource requirements are arbitrary, then no deterministic online algorithm can be competitive.

Consider once more a graph G with a single compute node and single data store node. However, this time the compute capacity will be 1 and the storage capacity will be some very large L. We will use up to $k+1$ jobs, each requiring $1/k$ units of computing. The i-th job, $0 \leq i \leq k$, will require the appropriate amount of storage so that its profit is M^i for very large M. Assuming $L = O(kM^k)$, all these storage requirements are at most $1/k$ of L. Note that storage can accommodate all jobs, but computing can accommodate at most k jobs. Any deterministic algorithm will have competitive ratio $\Omega(M)$ and k, M and L can be suitably large.

Thus, it follows that some flavor of Assumptions 1 and 2 are necessary to achieve any interesting competitive result.

Algorithm 1 Online algorithm for MaxCP.

$\lambda_u(1) \leftarrow 0, \lambda_v(1) \leftarrow 0$ for all $u \in U, v \in V$
for each new job j **do**
  $\text{cost}_u(j) \leftarrow \frac{1}{2}(e^{\lambda_u(j)\frac{\ln(2F+1)}{1-\epsilon}} - 1)$
  $\text{cost}_v(j) \leftarrow \frac{1}{2}(e^{\lambda_v(j)\frac{\ln(2F+1)}{1-\epsilon}} - 1)$
  For all uv let $Z_{juv} = \frac{p_j(u,v)}{c_u}\text{cost}_u(j) + \frac{s_j(u,v)}{d_v}\text{cost}_v(j)$
  Let uv maximize $f_j(u,v)$ subject to $Z_{juv} < f_j(u,v)$
  if such uv exist with $f_j(u,v) > 0$ **then**
    Assign j to uv
    $\lambda_u(j+1) \leftarrow \lambda_u(j) + \frac{p_j(u,v)}{c_u}$
    $\lambda_v(j+1) \leftarrow \lambda_v(j) + \frac{s_j(u,v)}{d_v}$
    For all other $u' \neq u$ let $\lambda_{u'}(j+1) \leftarrow \lambda_{u'}(j)$
    For all other $v' \neq v$ let $\lambda_{v'}(j+1) \leftarrow \lambda_{v'}(j)$
  else
    Reject job j
    For all u let $\lambda_u(j+1) \leftarrow \lambda_u(j)$
    For all v let $\lambda_v(j+1) \leftarrow \lambda_v(j)$
  end if
end for

We now present an $O(\log F)$-competitive and an $O(\log(kF))$-competitive algorithm for online MaxCP and Max$k$SP, respectively. Moreover, the lower bounds established in [5, Theorem 4.2] and [5, Theorem 4.3] yield matching lower bounds to online MaxCP and Max$k$SP, respectively.

Theorem 7 *There exists a deterministic $O(\log F)$-competitive algorithm for online* MaxCP *under Assumptions 1 and 2. For* Max$k$SP*, this can be extended to a $O(\log(kF))$-competitive algorithm.*

We establish the upper bound in Theorem 7 by adapting the framework of [5]. This framework uses an exponential cost function to place a price on remaining capacity of a node. If the profit obtained from a job can cover the cost of the capacity it consumes, we admit the job.

Our algorithm is given in Algorithm 1. In the description, e is the base of the natural logarithm. We first show that our algorithm will not exceed capacities. Essentially, this occurs because the cost will always be sufficiently high.

Lemma 10 *Capacity constraints are not violated at any time in Algorithm 1.*

Proof Note that $\lambda_u(n+1)$ will be $\frac{1}{c_u}\Sigma_{t,v}p_t(u,v)x_{tuv}$, since any time we assign a job j to uv we immediately increase $\lambda_u(j+1)$ by the appropriate amount. Thus if we can prove $\lambda_u(n+1) \leq 1$ we will not violate capacity of u.

Initially we had $\lambda_u(1) = 0 < 1$, so suppose that the first time we exceed capacity is after the placement of job j. Thus we have $\lambda_u(j) \leq 1 < \lambda_u(j+1)$. By applying assumption 2 we have $\lambda_u(j) > 1 - \epsilon$. From this it follows that $\text{cost}_u(j) > \frac{1}{2}(e^{\ln(2F+1)} - 1) = F$, and since these costs are always non-negative we must have had $Z_{juv} > \frac{p_j(u,v)}{c_u}F \geq f_j(u,v)$ by applying assumption 1. But then we must have rejected job j and would have $\lambda_u(j+1) = \lambda_u(j)$

An identical reasoning applies to $v \in V$.

Next, we bound the profit of our algorithm from below using the sum of the node costs.

Lemma 11 *Let $\mathcal{A}(j)$ be the total objective value $\Sigma_{t,u,v} x_{tuv} f_t(u,v)$ obtained by the algorithm immediately before job j arrives. Then $(3e\ln(2F+1))\mathcal{A}(j) \geq \sum_{u\in U} cost_u(j) + \sum_{v\in V} cost_v(j)$.*

Proof The proof will be by induction on j; the base case where $j = 1$ is immediate since no jobs have yet arrived or been scheduled and $\text{cost}_u(1) = \text{cost}_v(1) = 0$ for all u and v.

Consider what happens when job j arrives. If this job is rejected, neither side of the inequality changes and the induction holds. Otherwise, suppose job j is assigned to uv. We have:

$$\mathcal{A}(j+1) = \mathcal{A}(j) + f_j(u,v)$$

We can bound the new value of the righthand side by observing that since cost_u has derivative increasing in the value of λ_u, the new value will be at most the new derivative times the increase in λ_u. It follows that:

$$\begin{aligned}
\text{cost}_u(j+1) &\leq \text{cost}_u(j) + \frac{1}{2}(\lambda_u(j+1) - \lambda_u(j)) \left(\frac{\ln(2F+1)}{1-\epsilon}\right) e^{\frac{\lambda_u(j+1)\ln(2F+1)}{1-\epsilon}} \\
&= \text{cost}_u(j) + \frac{1}{2}\frac{p_j(u,v)}{c_u}\frac{\ln(2F+1)}{1-\epsilon} e^{\frac{\lambda_u(j)\ln(2F+1)}{1-\epsilon}} e^{\frac{p_j(u,v)\ln(2F+1)}{c_u(1-\epsilon)}} \\
&\leq \text{cost}_u(j) + \frac{1}{2}\frac{p_j(u,v)}{c_u}\frac{\ln(2F+1)}{1-\epsilon} e^{\frac{\lambda_u(j)\ln(2F+1)}{1-\epsilon}} e^{\frac{\epsilon\ln(2F+1)}{1-\epsilon}} \\
&\leq \text{cost}_u(j) + \frac{ep_j(u,v)}{c_u}\frac{\ln(2F+1)}{1-\epsilon}\left(\text{cost}_u(j) + \frac{1}{2}\right) \\
&\leq \text{cost}_u(j) + 2e\ln(2F+1)\left(\frac{p_j(u,v)}{c_u}\text{cost}_u(j) + \frac{1}{4}\right)
\end{aligned}$$

For the second step, we use the fact that $\lambda_u(j+1) = \lambda_u(j) + p_j(u,v)/c_u$. For the third step, we apply $p_j(u,v) \leq \epsilon c_u$ from Assumption 2. For the fourth step, we use the definition of $\text{cost}_u(j)$ and the fact that $\frac{\epsilon}{1-\epsilon} \leq \frac{1}{\ln(2F+1)}$. For the final step, we again invoke $p_j(u,v) \leq \epsilon c_u$ from Assumption 2 and note that $\epsilon \leq 1/2$.

An identical reasoning can be applied to cost_v, allowing us to show that the increase in the righthand side is at most:

$$2e\ln(2F+1)\left(\frac{p_j(u,v)}{c_u}\text{cost}_u(j) + \frac{s_j(u,v)}{d_u}\text{cost}_v(j) + \frac{1}{2}\right)$$

Since j was assigned to uv, we must have $f_j(u,v) > \frac{p_j(u,v)}{c_u}\text{cost}_u(j) + \frac{s_j(u,v)}{d_v}\text{cost}_v(j)$; from assumption 1 we also have $f_j(u,v) \geq 1$ so we can conclude that the increase in the righthand side is at most:

$$(3e\ln(2F+1))f_j(u,v) \leq (3e\ln(2F+1))(\mathcal{A}(j+1)-\mathcal{A}(j))$$

Now, we can bound the profit the optimum solution gets from jobs which we either fail to assign, or assign with a lower value of $f_t(u,v)$. The reason we did not assign these jobs was because the node costs were suitably high. Thus, we can bound the profit of jobs using the node costs.

Lemma 12 *Suppose the optimum solution assigned j to u,v, but the online algorithm either rejected j or assigned it to some u',v' with $f_j(u',v') < f_j(u,v)$. Then $\frac{p_j(u,v)}{c_u}cost_u(n+1) + \frac{s_j(u,v)}{d_v}cost_v(n+1) \geq f_j(u,v)$*

Proof When the algorithm considered j, it would find the u,v with maximum $f_j(u,v)$ satisfying $Z_{juv} < f_j(u,v)$. Since the algorithm either could not find such u,v or else selected u',v' with $f_j(u',v') < f_j(u,v)$ it must be that $Z_{juv} \geq f_j(u,v)$. The lemma then follows by inserting the definition of Z_{juv} and then observing that cost_u and cost_v only increase as the algorithm continues.

Lemma 13 *Let $\mathcal{Q}$ be the total profit of jobs which the optimum offline algorithm assigns, but which Algorithm 1 either rejects or assigns to a uv with lower profit of $f_t(u,v)$. Then $\mathcal{Q} \leq \Sigma_{u\in U}cost_u(n+1) + \Sigma_{v\in V}cost_v(n+1)$.*

Proof Consider any job q as described above. Suppose offline optimum assigns q to u_q, v_q. By applying lemma 12 we have:

$$\mathcal{Q} = \Sigma_q f_q(u_q,v_q) \leq \Sigma_q \frac{p_q(u_q,v_q)}{c_u}\text{cost}_{u_q}(n+1) + \frac{s_q(u_q,v_q)}{d_v}\text{cost}_q(n+1)$$

The lemma then follows from the fact that the offline algorithm must obey the capacity constraints.

Finally, we can combine Lemmas 11 and 13 to bound our total profit. In particular, this shows that we are within a factor $1 + 3e\ln(2F+1)$ of the optimum offline solution, where the additive unit factor accounts for the total profit that the optimum solution obtains from jobs j assigned to (u,v) by the optimum and (u',v') by the algorithm such that $f_j(u,v) \leq f_j(u',v')$. We have an $O(\log F)$-competitive algorithm, thus completing the proof of Theorem 7.

We can extend the result to k-sided placement, and can get a slight improvement in the required assumptions if we are willing to randomize. The results are given below:

Theorem 8 *For the k-sided placement problem, we can adapt algorithm 1 to be $O(\log(kF))$-competitive provided that assumption 2 is tightened to $\epsilon = \min(\frac{1}{2}, \frac{1}{1+\ln(kF+1)})$.*

Proof We modify the definition of cost to the following.

$$\text{cost}_u(j) = \frac{1}{k}\left(e^{\frac{\lambda_u(j)\ln(kF+1)}{1-\epsilon}} - 1\right)$$

.

The rest of the proof follows the $k = 2$ case. The competitive ratio increases to $O(\log(kF))$ because we need to assign the first job to arrive (otherwise after this job our competitive ratio would be unbounded). This job potentially uses up space on k machines while obtaining a profit of only 1. So as k increases, the ratio of "best" to "worst" job increases as well.

Theorem 9 *If we select $z \in [1, F]$ to be a random power of two and then reject all placements with $f_t(u, v) < z$ or $f_t(u, v) > 2z$, then we can obtain a competitive ratio of $O(\log F \log k)$ while weakening assumption 2 to $\epsilon = \min(\frac{1}{2}, \frac{1}{\ln(2k+1)})$. Note that in the specific case of two-sided placement this is $O(\log F)$-competitive requiring only that no single job consumes more than a constant fraction of any machine.*

Proof Once we make our random selection of z, we effectively have $F = 2$ and can apply the algorithm and analysis above. The selection of z causes us to lose (in expectation) all but $\frac{1}{\log F}$ of the possible profit, so we have to multiply this into our competitive ratio.

5 Concluding remarks

We introduce minimization and maximization versions of the k-sided placement, a generalization of the generalized assignment problem (GAP). For the minimization version, Min$k$SP, we present a $k + 1$ approximation using iterative rounding, thus generalizing the 2-approximation result for the minimization version of GAP. The best lower bound on inapproximability for Min$k$SP is a constant factor, derived from GAP. Finding the best polynomial-time approximation achievable for Min$k$SP is an interesting open problem. We also show that the particular linear program we use for Min$k$SP has an integrality gap that grows as $\Omega(\log k / \log\log k)$. The maximization version of k-sided placement, Max$k$SP, can be approximated to within a factor of $O(k)$ by applying randomized rounding to a k-column sparse LP relaxation [8]. We present simpler combinatorial algorithms based on local search for Max$k$SP and MaxCP (Max$k$SP with $k = 2$) that yield $O(k^2)$ and $15 + \epsilon$ approximations, respectively. Future research directions include developing combinatorial algorithms with better approximations and finding the best polynomial-time approximations achievable for the two problems.

Acknowledgments

We would like to thank Aravind Srinivasan for helpful discussions, and for pointing us to the $\Omega(k/\log k)$-hardness result for k-set packing, in particular.

We thank the anonymous referees for detailed insightful comments on earlier versions of the paper, and are especially grateful to a referee who generously offered a key insight leading to improved results for MinCP and Min$k$SP. We are thankful to Nikhil Bansal for pointing us to the recent results of Harris and Srinivasan on the Moser-Tardos framework for the Lovász Local Lemma, and their application to Min$k$SP. The third author was partly supported by awards NSF CNS-1217981 and CCF-1422715, and a grant from ONR.

References

1. Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *Transactions on Computer Systems*, 19:483–518, November 2001.
2. Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the Conference on File and Storage Technologies*, pages 175–188, 2002.
3. Eric Anderson, Mahesh Kallahalla, Susan Spence, Ram Swaminathan, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23:337–374, 2005.
4. K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing, and B. Rochwerger. Oceano-SLA based management of a computing utility. In *Proceedings of the International Symposium on Integrated Network Management*, pages 855–868, 2001.
5. Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput-competitive on-line routing. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 32–40, 1993.
6. Yossi Azar and Amir Epstein. Convex programming for scheduling unrelated parallel machines. In *In Proc. of the 37th Annual ACM Symposium on Theory of Computing (STOC 05*, pages 331–337, 2005.
7. N. Bansal, September 2014. Personal communication.
8. Nikhil Bansal, Nitish Korula, Viswanath Nagarajan, and Aravind Srinivasan. On k-column sparse packing programs. In *Proceedings of the Conference on Integer Programming and Combinatorial Optimization*, pages 369–382, 2010.
9. Niv Buchbinder and Joseph Naor. Improved bounds for online routing and packing via a primal-dual approach. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 293–304, 2006.
10. Niv Buchbinder and Joseph Naor. The design of competitive online algorithms via a primal-dual approach. *Foundations and Trends in Theoretical Computer Science*, 3(2-3):93–263, 2009.
11. Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the Symposium on Operating Systems Principles*, pages 103–116, 2001.
12. Chandra Chekuri and Sanjeev Khanna. A PTAS for the multiple knapsack problem. In *Proceedings of the Symposium on Discrete Algorithms*, pages 213–222, 2000.
13. Marek Cygan, Fabrizio Grandoni, and Monaldo Mastrolilli. How to sell hyperedges: The hypermatching assignment problem. In *SODA*, pages 342–351, 2013.
14. L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Surveys*, 14, 1982.
15. Lisa Fleischer, Michel X. Goemans, Vahab S. Mirrokni, and Maxim Sviridenko. Tight approximation algorithms for maximum general assignment problems. In *SODA*, pages 611–620, 2006.
16. D. Harris and A. Srinivasan. The moser-tardos framework with partial resampling, June 2014. arXiv:1406.5943.

17. David G. Harris and Aravind Srinivasan. Constraint satisfaction, packet routing, and the Lovász Local Lemma. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 685–694, 2013.
18. David G. Harris and Aravind Srinivasan. The moser-tardos framework with partial resampling. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 469–478, 2013.
19. Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *Computational Complexity*, 15(1):20–39, 2006.
20. Madhukar Korupolu, Aameek Singh, and Bhuvan Bamba. Coupled placement in modern data centers. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–12, 2009.
21. L.C. Lau, R. Ravi, and M. Singh. *Iterative Methods in Combinatorial Optimization*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2011.
22. Jan K. Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(3):259–271, 1990.
23. David A. Patterson. Technical perspective: the data center is the computer. *Communications of the ACM*, 51:105–105, January 2008.
24. David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(3):461–474, 1993.