Coupled and k-Sided Placements: Generalizing Generalized Assignment

Madhukar Korupolu[1], Adam Meyerson[1], Rajmohan Rajaraman[2], and Brian Tagiku[1]

[1] Google, 1600 Amphitheater Parkway, Mountain View, CA. Email: {mkar,awmeyerson,btagiku}@google.com
[2] Northeastern University, Boston, MA 02115. Email: rraj@ccs.neu.edu

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 $O(k)$ 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 approximation algorithm that yields a 15-approximation and $O(k^3)$-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 [1]. 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, compute 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 compute location is fixed [2, 3] or placing compute requirements assuming job storage location is fixed [4, 5]. 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 compute and storage elements can benefit by being placed on a storage node that has a nearby available compute 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 [6] 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 processing requirement $p_j : E \to \mathcal{R}$ and a storage requirement $s_j : E \to \mathcal{R}$. We note that without loss of generality, we can assume that the capacities are unit, since we can scale the processing and storage requirements of individual nodes accordingly.

We consider two versions of the coupled placement problems. For the maximization version MaxCP, we view f_j as a payment 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. 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 $\sum_{j \in A} f_j(\sigma(j))$ is minimized.

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.

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, it is easy to see that 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 next consider the maximization version. Max$k$SP can be expressed as a k-column sparse integer packing program (k-CSP). 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 [7]. An $\Omega(k/\log k)$-inapproximability result for k-set packing due to [16] 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 a local search based 15-approximation algorithm for MaxCP. We extend it to Max$k$SP and obtain an $O(k^3)$-approximation.

The local search result applies directly to a version where we can assign tasks fractionally but only to a single pair of machines (this is like assigning a task with lower priority and may have additional applications). 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 and k-sided placements assign jobs to tuples of machines.

Finally, we study the online version of MaxCP, in which tasks arrive online and must be irrevocably assigned or rejected immediately upon arrival.

– We extend the techniques of [8] to the case where the capacity requirement for a job is arbitrarily machine-dependent. This enables us to achieve competitive ratio logarithmic in the ratio of best to worst value-per-capacity density, under necessary technical assumptions about the maximum job size.

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 [9], the goal is 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 [9] give an algorithm for computing an assignment that achieves optimal cost while doubling the capacities of each bin. A previous result by Lenstra *et al.* [10] 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 [11] 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 [12]. It is known that MaxGAP is APX-hard [11], 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 compute location is fixed (file allocation problem [2], [13, 14, 3]) or placing compute requirements assuming job storage location is fixed [4, 5]. These in a sense are variants of GAP. The only prior work on Coupled Placement is [6], 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 Max$k$SP problem is related to the recently studied hypermatching assignment problem (HAP) [15], and special cases, including k-set packing, and a uniform version of the problem. A $(k + 1 + \varepsilon)$-approximation is given for HAP in [15], where other variants of HAP are also studied. While the Max$k$SP problem can be viewed as a variant of HAP, there are critical differences. For instance, in Max$k$SP, each task is assigned at most one tuple, while in the hypermatching problem each client (or task) is assigned a subset of the hyperedges. Hence, the Max$k$SP 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 [16] 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 (processing and storage). We consider the linear relaxation of this program which replaces the integrality constraints above with $0 \leq x_{tuv} \leq 1, \forall t \in T, u \in U, v \in V$.

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 task satisfaction constraints, computation constraints, and storage constraints, respectively, in x. Note that every task 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 $N = n_t + n_c + n_s$.

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 $n_s + n_c = N - n_t \geq N/2$. 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 tasks plus the capacity already used in earlier iterations. Since each task 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. Since each satisfaction constraint is tight, each variable appears in 2 capacity constraints, and each variable takes on value less than 1, the sum of all the variables over the 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 tasks 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 tasks be numbered 1 through ℓ. Let the demand of task 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:

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

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.

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

It is straightforward to generalize the the algorithm of the preceding section to obtain a $k+1$-approximation to Min$k$SP. We first set up the integer LP for Min$k$SP. 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.

The claims and proofs are almost identical to the $k = 2$ case and are moved to Appendix A. A natural question to ask is whether a linear approximation factor of Min$k$SP is unavoidable for polynomial time algorithms. Unfortunately, we do

not have any non-trivial results in this direction. We have been able to show that the MINkSP linear program has an integrality that grows as $\Omega(\log k / \log\log k)$ (see Appendix A).

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} denote the indicator variable for the the assignment of job t to the pair (u, v), $u \in U$, $v \in V$. The goal is then to

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

Note that we can deal with capacities on u, v by scaling the $p_t(u, v)$ and $s_t(u, v)$ values appropriately. The above LP can be easily extended to MAXkSP (see Appendix B). These linear programs are 3- and k-column sparse packing programs, respectively, and can be approximated to within a factor of 15.74 and $ek + o(k)$, respectively using a clever randomized rounding approach. We next give a combinatorial approach based on local search which is likely to be much more efficient in practice.

3.2 Approximation algorithms based on local search

Before giving the details, we start with a few helpful definitions. For any $u \in U$, $F_u = \Sigma_{t,v} x_{tuv} f_t(u, v)$. Similarly, for any $v \in V$, $F_v = \Sigma_{t,u} x_{tuv} f_t(u, v)$. We set $\mu = \frac{1}{n} \max_{t,u,v} f_t(u, v)$. It follows that the optimum solution is at least $n\mu$ and at most $n^2\mu$.

The local search algorithm will maintain the following two invariants: (1) For each t, there is at most one pair (u, v) for which $x_{tuv} > 0$; (2) All the linear program inequalities hold. It's easy to set an initial state where the invariant holds (all $x_{tuv} = 0$). The local search algorithm proceeds in the following steps: While $\exists t, u, v : f_t(u, v) > F_u \frac{p_t(u,v)}{c_u} + F_v \frac{s_t(u,v)}{d_v} + \Sigma_{u',v'} x_{tu'v'} f_t(u', v') + \epsilon\mu$:

1. Set $x_{tuv} = 1$ and set $x_{tu'v'} = 0$ for all $(u', v') \neq (u, v)$.
2. While $\Sigma_{t,v} p_t(u, v) x_{tuv} > c_u$, reduce x_{tuv} for the job with minimum $c_u f_t(u, v)/p_t(u, v)$ such that $x_{tuv} > 0$.
3. While $\Sigma_{u,v} s_t(u, v) x_{tuv} > d_v$, reduce x_{tuv} for the job with minimum $d_v f_t(u, v)/s_t(u, v)$ such that $x_{tuv} > 0$

Theorem 2. *The local search algorithm maintains the two stated invariants.*

Proof. The first invariant is straightforward, because the only time we increase an x_{tuv} value we simultaneously set all other values for the same t to zero. The only time the linear program inequalities can be violated is immediately after setting $x_{tuv} = 1$. However, the two steps immediately after this operation will reduce the values of other jobs so as to satisfy the inequalities (and this is done without increasing any x_{tuv} so no new constraint can be violated).

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

Proof. When the algorithm terminates, we have for all t, u, v: $f_t(u, v) \leq F_u \frac{p_t(u,v)}{c_u} + F_v \frac{s_t(u,v)}{d_v} + \Sigma_{u',v'} x_{tu'v'} f_t(u', v')\epsilon\mu$. We sum this over t, u, v representing the optimum integer assignments: $OPT \leq \Sigma_u F_u + \Sigma_v F_v + \Sigma_{t,u,v} x_{tuv} f_t(u, v) + \epsilon OPT$. Each summation simplifies to the algorithm's objective value, giving the result.

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

Proof. Setting $x_{tuv} = 1$ and setting all other $x_{tu'v'} = 0$ adds $f_t(u, v) - \Sigma_{u'v'} x_{tu'v'} f_t(u', v')$ to the algorithm's objective. The next two steps of the algorithm (making sure the LP inequalities hold) reduce the objective by at most $F_u \frac{p_t(u,v)}{c_u} + F_v \frac{s_t(u,v)}{d_v}$. It follows that each iteration of the main loop increases the solution value by at least $\epsilon\mu$. By definition of μ, this can happen at most n^2/ϵ times. Each selection of (t, u, v) can be done in polynomial time (at worst, by simply trying all tuples).

Rounding Phase: When the local search algorithm terminates, we have a fractional solution with the additional guarantee from the first invariant. Note that we can extend this to the k-sided version if we increase the approximation factor to $k+1+\epsilon$. Below, we give two different rounding schemes. The first works for general values of k and loses an $O(k^2)$ factor, for an overall approximation factor of $O(k^3)$. The second is specific to the $k = 2$ case and obtains a better approximation.

1. We randomly make each assignment with probability p times the fractional value (so px_{tuv} for Coupled Placement), for some p to be defined later.
2. For each assigned job t, if the other jobs $t' \neq t$ assigned to any one of its assigned machines violate the corresponding linear program constraint, we immediately drop job t. For Coupled Placement this means if $\sum_{t' \neq t, v} p_{t'}(u, v) x_{t'uv} > 1$ for any t, u we set $x_{tuv} = 0$.

3. Note that we may still violate linear program constraints, but for any particular machine the constraint would be satisfied if we dropped any one of its assigned jobs. We divide the assigned jobs into $k+1$ groups. These groups should guarantee that for any machine with at least two assigned jobs, not all its jobs are members of the same group. We then select the group with largest total objective value as our final solution.

Theorem 5. *For the k-sided version, the rounding scheme runs in poly-time and achieves an $O(k^2)$-approximation over the fractional approximation factor (so an overall factor of $O(k^3)$ using local search) for appropriate choice of p.*

Proof. The first two steps finish with a solution of value at least $p(1-p)^k$ times the optimum in expectation. This is because for any job t, the probability of placing this job in step one is exactly p times its fractional value. Consider any machine m where the job is assigned; the expected total size of the other jobs $t' \neq t$ assigned to this machine is at most pc_m and thus the probability that these other jobs exceed c_m is at most p. The probability that none of the k machines where t is assigned exceed capacity from other jobs will be at most $(1-p)^k$.

We may still violate constraints. Dividing into $k+1$ groups and picking the best gives a result which is at least $\frac{1}{k+1}p(1-p)^k$ times optimum without violating constraints. Selecting $p = \frac{1}{k}$ gives the desired approximation factor.

It remains to show that the division into groups can be performed in poly-time. We start with all machines unmarked. For each group, we select a maximal set of jobs no two of which are assigned the same unmarked machine. We then mark all machines to which one of our current group of jobs is assigned. Note that immediately before we select group i, each remaining job is assigned to at most $k-i+1$ unmarked machines. For $i = 1$ this is obvious. Inductively, suppose that job j is assigned to more than $k-i$ unmarked machines immediately before selecting group $i+1$. Before selecting group i, job j was assigned to at most $k-i+1$ unmarked machines, and since we never "unmark" a machine it follows that job j was assigned to exactly $k-i+1$ unmarked machines both before and after the selection of group i. But then none of the jobs selected in group i are assigned to any of the unmarked machines assigned to job j (else they would have become marked after selection of group i). So we can augment group i with job j without violating the constraint that no two jobs of group i are on the same unmarked machine. This contradicts the maximality of group i.

We thus conclude that immediately before we select group $k+1$, each remaining job is assigned only to marked machines. Thus group $k+1$ selects all remaining jobs (maximality) and the jobs are divided into $k+1$ groups. Consider any machine m with at least two assigned jobs. Let group i be the first group to contain a job from m. Thus prior to selection of group i, we had not selected any job which was assigned to m and m was unmarked. So group i cannot include more than one job from machine m without violating the condition that no two jobs share an unmarked machine. It follows that there are at least two distinct groups which contain jobs from machine m (group i and also some later group).

For MaxCP, we can improve the approximation factor. We refer the reader to Appendix B for details.

Theorem 6. *For* MaxCP*, there exists a polynomial-time algorithm based on local search that achieves a* $15 + \epsilon$ *approximation for* MaxCP*.*

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 value at the end of the instance. We apply the techniques of [8] to obtain a logarithmic competitive online algorithm under certain assumptions. We first note that online MaxCP differs from the model considered in [8] in that a job's computation/storage requirements need not be the same.

As demonstrated in [8] 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 *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 *For* $\epsilon = \min(\frac{1}{2}, \frac{1}{\ln 2F+1})$*, 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 (proof in Appendix C).

Theorem 7. *No deterministic online algorithm 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 values and resource requirements are completely uncorrelated.*

A small modification to the algorithm of [8] gives an $O(\log F)$-competitive algorithm. Moreover, the lower bound of $\Omega(\log F)$ shown in [8] applies to online MaxCP as well. (See Appendix D for proof.)

Theorem 8. *There exists a deterministic* $O(\log F)$*-competitive algorithm for online* MaxCP *under Assumptions 1 and 2. For* Max*kSP, this can be extended to a* $O(\log kF)$*-competitive algorithm. Moreover, any online deterministic algorithm for online* MaxCP *has competitive ratio* $\Omega(\log F)$*, and for online* Max*kSP has competitive ratio* $\Omega(\log kF)$*.*

Theorem 9. *There exist a randomized* $O(\log F)$*-competitive algorithm (in expectation) for online* MaxCP *under assumption 1 even if we weaken assumption 2 to require only that* $\epsilon = \frac{1}{2}$*. No deterministic online algorithm for the problem can accomplish such a result.*

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 anonymous referees for helpful comments on an earlier version of the paper, and are especially grateful to a referee who generously offered the key insights leading to improved results for MinCP and Min$k$SP.

References

1. Patterson, D.A.: Technical perspective: the data center is the computer. Communications of the ACM **51** (January 2008) 105–105
2. Dowdy, L.W., Foster, D.V.: Comparative models of the file assignment problem. ACM Surveys **14** (1982)
3. Anderson, E., Kallahalla, M., Spence, S., Swaminathan, R., Wang, Q.: Quickly finding near-optimal storage designs. ACM Transactions on Computer Systems **23** (2005) 337–374
4. Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J., Rochwerger, B.: Oceano-SLA based management of a computing utility. In: Proceedings of the International Symposium on Integrated Network Management. (2001) 855–868
5. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., Doyle, R.P.: Managing energy and server resources in hosting centers. In: Proceedings of the Symposium on Operating Systems Principles. (2001) 103–116
6. Korupolu, M., Singh, A., Bamba, B.: Coupled placement in modern data centers. In: Proceedings of the International Parallel and Distributed Processing Symposium. (2009) 1–12
7. Bansal, N., Korula, N., Nagarajan, V., Srinivasan, A.: On k-column sparse packing programs. In: Proceedings of the Conference on Integer Programming and Combinatorial Optimization. (2010) 369–382
8. Awerbuch, B., Azar, Y., Plotkin, S.: Throughput-competitive on-line routing. In: Proceedings of the Symposium on Foundations of Computer Science. (1993) 32–40
9. Shmoys, D.B., Éva Tardos: An approximation algorithm for the generalized assignment problem. Mathematical Programming **62**(3) (1993) 461–474
10. Lenstra, J.K., Shmoys, D.B., Éva Tardos: Approximation algorithms for scheduling unrelated parallel machines. Mathematical Programming **46**(3) (1990) 259–271
11. Chekuri, C., Khanna, S.: A PTAS for the multiple knapsack problem. In: Proceedings of the Symposium on Discrete Algorithms. (2000) 213–222
12. Fleischer, L., Goemans, M.X., Mirrokni, V.S., Sviridenko, M.: Tight approximation algorithms for maximum general assignment problems. In: SODA. (2006) 611–620
13. Alvarez, G.A., Borowsky, E., Go, S., Romer, T.H., Becker-Szendy, R., Golding, R., Merchant, A., Spasojevic, M., Veitch, A., Wilkes, J.: Minerva: An automated resource provisioning tool for large-scale storage systems. Transactions on Computer Systems **19** (November 2001) 483–518
14. Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., Veitch, A.: Hippodrome: Running circles around storage administration. In: Proceedings of the Conference on File and Storage Technologies. (2002) 175–188
15. Cygan, M., Grandoni, F., Mastrolilli, M.: How to sell hyperedges: The hypermatching assignment problem. In: SODA. (2013) 342–351

16. Hazan, E., Safra, S., Schwartz, O.: On the complexity of approximating k-set packing. Computational Complexity **15**(1) (2006) 20–39
17. Vazirani, V.V.: Approximation Algorithms. Springer-Verlag (2001)
18. Frieze, A.M., Clarke, M.: Approximation algorithms for the m-dimensional 0-1 knapsack problem: Worst-case and probabilistic analyses. European Journal of Operational Research **15**(1) (1984) 100–109
19. Chekuri, C., Khanna, S.: On multi-dimensional packing problems. In: Proceedings of the Symposium on Discrete Algorithms. (1999) 185–194
20. Srinivasan, A.: Improved approximations of packing and covering problems. In: Proceedings of the Symposium on Theory of Computing. (1995) 268–276
21. Lau, L., Ravi, R., Singh, M.: Iterative Methods in Combinatorial Optimization. Cambridge Texts in Applied Mathematics. Cambridge University Press (2011)

A Proofs for Min$k$SP

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. Since each satisfaction constraint is tight, each variable appears in k capacity constraints, and each variable takes on value at most 1, the sum of all the variables over the 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 tasks 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 tasks be numbered 1 through ℓ. Let the demand of task 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 10. ITERROUND*(k) is a polynomial-time $k+1$-approximation algorithm for* MINk*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.

We now show that the MINkSP LP has an integrality gap of $\Omega(\log k / \log\log k)$. We recursively construct an integrality gap instance with ℓ^t sides, for parameters ℓ and t, with two nodes per side one with infinite capacity and the other with unit capacity, such that any integral solution has at least t tasks 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 $\ell = \Theta(\log k / \log\log k)$.

Each task can be placed on one tuple from a subset of tuples; for a given tuple, the demand of the task on each side of the tuple is one. We start with the construction for $t = 1$. We introduce a task 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, any integral solution uses up unit capacity of one unit-capacity node, while there is a fractional solution ($1/\ell$ for each choice) that uses only $1/\ell$ fraction of each unit capacity node.

Given a construction for ℓ^t sides, we show how to extend to ℓ^{t+1} sides. We take ℓ identical copies of the instance with ℓ^t sides and combine the tuples for each task in such a way that for any i, any integral placement places exactly the same task on side i of each copy. Now we add task $t + 1$ 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 i. Clearly, any integral solution will have to add one more task to a unit-capacity node of a side that already has load t, yielding a load $t + 1$, while a fractional solution can assign load of at most $1/\ell$ to the unit-capacity nodes of each side.

B Proofs for MAXkSP and MAXCP

We first present the linear program for MAXkSP (recall the definition in Section 1.1). Let x_{te} denote the indicator variable for the assignment of job t to the k-tuple e.

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

We now present the improved approximation algorithm for MAXCP. The idea is to obtain a one-to-one correspondance between fractional assignments and machines. Essentially we view the machines as nodes of a graph where the edges are the fractional assignments (this is similar to the rounding for generalized assignment). If we have a cycle, the idea is to shift the fractions around the cycle (i.e. increase one x_{tuv} then decrease some $x_{t'vw}$ and increase some $x_{t''wx}$ and so forth). Applying this directly on a single cycle may violate some constraints; while we try to increase and decrease the fractions in such a way that constraints hold, since each job has different "size" on its two endpoints we may wind up violating the constraint $\sum_{t,v} x_{tuv} p_t(u, v)$ at a single node u. This prevents us from doing a simple cycle elimination as in generalized assignment. However, if we have two adjoining (or connected) cycles the process can be made to work. The remaining case is a single cycle, where we can assign each edge to one of its endpoints. Generalized assignment rounding would now proceed to integrally assign each job to its corresponding machine; we cannot do this because each job requires *two* machines, and each machine thus has multiple fractional assignments (all but one of which "correspond" to some other machine).

Lemma 7. *Given any fractional solution which satisfies the local search invariants, we can produce an alternative fractional solution (also satisfying the local search invariants and with equal or greater value). This new fractional solution labels each job t with $0 < x_{tuv} < 1$, with either u or v, guaranteeing that each u is labeled with at most one job.*

Proof. Consider a graph where the nodes are machines, and we have an edge (u, v) for any fractional assignment $0 < x_{tuv} < 1$. If any node has degree zero or one, we remove that node and its assigned edge (if any), labeling the removed edge with the node that removed it. We continue this process until all remaining nodes have degree at least two. If there is a node of degree three, then there must exist two (distinct but not necessarily edge-disjoint) cycles with a path between them (possibly a path of length zero); since the graph is bipartite all cycles are even in length. We can alternately increase and decrease the fractional assignments of edges along a cycle such that the total load $\sum_{t,v} p_t(u, v) x_{tuv}$ changes

only on a single node u where the path between cycles intersects this cycle. We can do the same along the other cycle. We can then do the same thing along the path, and equalize the changes (multiplicatively) such that there is no overall change in load, but at least one edge has its fractional value changing. If this process decreases the value, we can reverse it to increase the value. This allows us to modify the fractional solution in a way that increases the number of integral assignments without decreasing the value. After applying this repeatedly (and repeating the node/edge removal process above where necessary), we are left with a graph that consists only of node-disjoint cycles. Each of the remaining edges will be labeled with one of its two endpoints (one to each). The overall effect is that we have a one-to-one labeling correspondance between fractional assignments and machines (each fractional edge to one of its two assigned machines). Note however that since each job is assigned to two machines and labeled with only one of the two, this does not imply that each machine has only one fractional assignment.

Once this is done, we consider three possible solutions. One consists of all the integral assignments. The second considers only those assignments which are fractional and labeled with nodes u. For each node v, we select a subset of its fractional assignments to make integrally, so as to maximize the value without violating capacity of v. We cannot violate capacity of u because we select at most one job for each such machine. The result has at least $\frac{1}{2}$ the value of assignments labeled with nodes u. For the third solution, we do the same but with the roles of u, v reversed. We select the best of these three solutions; our choice obtains at least $\frac{1}{5}$ of the overall value.

Proof of Theorem 6: The algorithm sketch contains most of the proof. We need to establish that we can get at least $\frac{1}{2}$ the fractional value on a single machine integrally. This can be done by selecting jobs in decreasing order of density $(f_t(u,v)/p_t(u,v))$ until we overflow the capacity. Including the job that overflows capacity, this must be better than the fractional solution. Thus we can select either everything but the job that overflows capacity, or that job by itself.

We also need to establish the $\frac{1}{5}$ value claim. If we were to select the integral assignments with probability $\frac{1}{5}$ and each of the other two solutions with probability $\frac{2}{5}$, we would get an expected $\frac{1}{5}$ of the fractional solution. Deterministially selecting the best of the three solutions can only be better than this. □

C Proof of Theorem 7

We first show that if resource requirements are large compared to capacities, payment functions f_t are exactly equal to the total amount of resources and each job requires the same amount over all resources/dimensions (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 one-dimensional 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, payment 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 one-dimensional, unit capacities. 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 value 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 at least $\Omega(M)$ and M and k can be arbitrarily large.

Similarly, if resource requirements are small relative to capacities, payment 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 with one-dimensional compute/storage capacities. 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 value 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.

D Proof of Theorem 8

We adapt the framework of [8] to solve the online MaxCP problem. This framework uses an exponential cost function to place a price on remaining capacity of a node. If the value obtained from a task can cover the cost of the capacity it consumes, we admit the task. In the algorithm below, 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 8. *Capacity constraints are not violated at any time during this algorithm.*

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 task 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 $Z_{tuv} = \frac{p_j(u,v)}{c_u}\text{cost}_u(j) + \frac{s_j(u,v)}{d_v}\text{cost}_v(j)$
  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 task 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

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 appopriate 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 $\lambda_u(j+1) = \lambda_u(j)$

Identical reasoning applies to $v \in V$.

Next, we bound the algorithms revenue from below using the sum of the node costs.

Lemma 9. *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:

$$\text{cost}_u(j+1) \leq \text{cost}_u(j) + (\lambda_u(j+1) - \lambda_u(j))\frac{1}{2}(\frac{\ln(2F+1)}{1-\epsilon})(e^{\frac{\lambda_u(j+1)\ln(2F+1)}{1-\epsilon}})$$

$$\text{cost}_u(j+1) \leq \text{cost}_u(j) + \frac{p_j(u,v)}{c_u}(\frac{\ln(2F+1)}{1-\epsilon})(\frac{1}{2}e^{\frac{\lambda_u(j)\ln(2F+1)}{1-\epsilon}})(e^{\frac{\epsilon\ln(2F+1)}{1-\epsilon}})$$

$$\text{cost}_u(j+1) \leq \text{cost}_u(j) + \frac{p_j(u,v)}{c_u}\frac{\ln(2F+1)}{1-\epsilon}(\text{cost}_u(j) + \frac{1}{2})(e^{\epsilon\ln(2F+1)})$$

Applying assumption 2 gives:

$$\text{cost}_u(j+1) \leq \text{cost}_u(j) + (2e\ln(2F+1))(\frac{p_j(u,v)}{c_u}cost_u(j) + \frac{1}{4})$$

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

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

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

Lemma 10. *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 algortihm 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 11. *Let $\mathcal{Q}$ be the total value of tasks which the optimum offline algorithm assigns, but which Algorithm 1 either rejects or assigns to a uv with lower value 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 task q as described above. Suppose offline optimum assigns q to u_q, v_q. By applying lemma 10 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 9 and 11 to bound our total profit. In particular, this shows that we are within a factor $3e \ln(2F+1)$ of the optimum offline solution, for an $O(\log F)$-competitive algorithm.

Theorem 11. *Algorithm 1 never violates capacity constraints and is $O(\log F)$-competitive.*

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

Proof. We must modify the definition of cost to:

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

The rest of the proof will then go through. The intuition for the increase in competitive ratio is that we need to assign the first task to arrive (otherwise after this task our competitive ratio would be unbounded). This task potentially uses up space on k machines while obtaining a value of only 1. So as the value of k increases, the ratio of "best" to "worst" task effectively increases as well.

Theorem 13. *If we select a random power of two $z \in [1, F]$ 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.