Repairing Bin Packing Constraints (Extended Abstract)

Alex S. Fukunaga

Global Edge Institute, Tokyo Institute of Technology, Meguro, Tokyo, Japan
fukunaga@is.titech.ac.jp,

Abstract. We consider a variant of the bin packing problem, where items are initially assigned to bins, and the goal is to rearrange the items in the minimum number of steps such that the capacity constraints are satisfied. We consider two search spaces for solving this problem, a commitment-based search space and a difference-based search space. We evaluate depth-first branch and bound and IDA* algorithms in these search spaces, and show that IDA* in the commitment-based search space significantly outperforms the alternatives.

1 Introduction: The Min-Cost Load Rebalancing Problem

Consider a set of m *servers* (e.g., web servers) each with capacity $c_1, ..., c_m$, and given a set of n *jobs*, where each job has a weight $w_1, ..., w_n$ representing the amount of workload that the job demands of a server. For example, each job could represent a web site that is located at a particular server. The load balancing problem of determining whether each job can be assigned to exactly one server such that the sum of all jobs assigned to each bin j is less than or equal to c_j is an instance of the decision version of the classical NP-complete *bin packing problem*. [5] Note that in this web server example, exceeding C would cause the overloaded server to fail or have unacceptable response times.

Now, suppose that due to changes in site popularity or content (e.g., increased downloads of bandwidth-intensive media files), the current loads on the servers differ from loads that were projected when the initial server assignments were made. Some of the servers are becoming overloaded. It is possible to reassign jobs to servers in order to rebalance the loads. However, moving a job between servers incurs costs (direct costs include system administration costs, intangible costs are the costs of downtime for a web service). The *min-cost load rebalancing problem* is the problem of finding a new assignment of jobs to servers such that (1) no server is overloaded, and (2) the number of jobs that are moved between severs is minimized. A closely related problem was considered in [1], which presented approximation algorithm for the problem of minimizing the maximum load on any server while moving less than k jobs. Similar problems also exist in the problem of process migration in distributed systems.

More formally, the Min-Cost Load Rebalancing Problem (MCLRP) is defined as follows: Given m servers with capacities $c_1, ..., c_m$, a set of n jobs with

weights $w_1, ..., w_n$, and an initial assignment $I_1, ..., I_n$ of jobs to servers, find a new assignment of jobs to servers $x_1, ..., x_n$, $1 \leq x_i \leq m$, such that each job is assigned to exactly one server, and for every server j, the sum of the job weights assigned to it is less than or equal to its capacity c_j. The objective is to minimize $\sum_{i=1}^{n}(x_i \neq I_i)$, the number of differences between the initial and final assignments. The MCLRP is NP-hard, by reduction from bin packing.

2 Algorithms for the MCLRP

One possible approach to solving the MCLRP is to modify an existing bin packing algorithm such that instead of terminating after finding a feasible bin assignment of k bins, it searches for a k-bin solution with minimal distance from the initial assignment. However, this is difficult because the optimal solution to the MCLRP can be pruned by the techniques used by bin packing solvers. All of the currently available bin packing algorithms, including column generation solvers such as [2], search-based methods [4], as well as classical algorithms such as the Martello-Toth procedure [9], all derive their effectiveness from mechanisms such as column generation, lower bounds, and dominance detection, which, if applied straightforwardly, can prune the optimal MCLRP solution. In other words, it is nontrivial to start with an existing bin packing solver and derive an algorithm for finding optimal solutions the MCLRP without disabling or substantially altering the pruning mechanisms that are responsible for making the bin packing solver effective. El Sakkout and Wallace considered a minimal cost repair problem for an abstract scheduling problem. [3], and proposed a general approach, probe backtracking, for addressing min-perturbation problems in CSPs. A key difference from the MCLRP is that they consider difference functions that can be expressed linearly (the MCLRP difference count is not expressible in their model). Also, their probe backtracking algorithm does not explicitly consider the initial schedule, and reschedules from scratch.

In this paper, we focus on exact (complete) search algorithms for the MCLRP. Given an initial state $X_0 = x_{1,0}..., x_{n,0}$, let D_i be the set of states which differ from X_0 by exactly i variable assignment (i.e., i variables in $X \in D_i$ have a value which is different from their value in the initial state X_0). Consider the set $D = D_1 \cup D_2 \cup ...D_n$. We call D the *difference space*, or *D-space*.

We can perform a standard depth-first branch-and-bound (DFBNB) search in D-space, where at each node, we select a variable x and assign it some value which differs from the value in the initial state X_0. The lower bounds and infeasibility checks described below are applied at each node.

Another way to view the MCLRP is as a path-finding problem, where the start state is the initial assignment, and the objective is to find a goal state with minimal distance from the start state. Thus, we consider IDA* [8], which expands nodes in a best-first order using linear space (at the cost of reopening some nodes). The admissible heuristic function used by IDA* is the same as the lower bounding function used for DFBNB (see below), and the d-th iteration of IDA* explores the subset of the DFBNB D-space search tree where at each

node, the sum $f = g + h \leq d$, where g is the number of differences from the initial state in the current solution, and h is the lower bound on the additional number of differences required to find a conflict-free solution.

Search in D-space has been the basis of previous work on finding minimal perturbation solutions from some initial state for constraint satisfaction problems. Ran et al have applied IDA* in D-space to solve a minimal perturbation problem for binary CSPs [11]. Verfaillie and Schiex applied a depth-first backtracking algorithm to solve dynamic CSPs [12].

An alternative search space is a *commitment-based search space* (C-space), where each node in the search tree represents a partially committed assignment of variables to values. That is, each variable is assigned some value, as well as whether a commitment has been made to the value. We say that an variable x is *committed* to a value v at a search node N if in at N and every descendant of the node, x is assigned to v, and *uncommitted* otherwise. For variables $x_1, ..., x_n$, we denote a search state as the list $S = \{x_1 = val_1, ..., x_n = val_n\}$, or more concisely, $\{val_1, ..., val_n\}$. Furthermore, the values are annotated with an underline "_" if the variable is committed to that value. For example, in a 2-variable MCRP where the current assignments are $v_1 = 1, v_2 = 2$, and we have committed $v_1 = 1$, we can denote this state as $\{\underline{v_1 = 1}, v_2 = 2\}$, or more concisely, $\{\underline{1}, 2\}$. Initially, the variables are assigned the values of the initial assignment I, and all variables are uncommitted.

As with D-space, it is possible to apply either a depth-first branch-and-bound or an IDA* search strategy in C-space. DFBNB in C-space was previously proposed by Minton et al [10]. IDA* in C-space is clearly related to Limited Discrepancy search (LDS) [6], as both algorithms search a space which is limited by some notion of "discrepancy". In fact, it has been noted that LDS can be viewed as a best-first search, where the cost of a node is the number of discrepancies in its path from the root [7]. In LDS, a "discrepancy" refers to a decision which deviates from the first value suggested by a value-ordering heuristic. Let δ be the class of all value ordering heuristics where the first value suggested for variable x_i by the value ordering is the value of x_i in the initial state. Using some value ordering from the class δ, we can implement LDS in C-space which is similar to IDA* in C-space. The differences are: (1) On the d-th iteration, LDS explores nodes with up to d discrepancies. On the other hand, on the d-th iteration, IDA* explores all nodes where at each node, the sum $f = g + h \leq d$, where g is the number of discrepancies so far, and h is a lower bound on the additional number of discrepancies required before a conflict-free state is reached. (2) IDA*, like DFBNB, is a strategy which specifies the overall backtracking strategy. This is orthogonal to the selection of a *value ordering strategy*, which specifies the order in which children of a node are sorted. LDS and its variants prescribe both a backtracking strategy as well as a particular value ordering strategy (e.g., in the context of the MCLRP, LDS would use a strategy from the class δ). Thus, IDA* in C-space is a generalization of LDS for the MCLRP, i.e., LDS for the MCLRP is a special case of the C-space IDA* with a trivial lower bound, $h = 0$, and a value ordering heuristic from class δ.

We have been evaluating a number of variable ordering and value ordering heuristics in both C-space and D-space. In our experiments, we used a standard most-constrained variable (item) ordering, and a simple, random value (bin) ordering for all combinations of C-space, D-space, DFBNB, and IDA*. Results using different variable and value ordering strategies are qualitatively similar.

D-space can be defined in terms of C-space as the subset of C-space where all committed variables are assigned a value that is different than the initial assignment. For example, given variables v_1, v_2 and initial assignment $v_1 = 1, v_2 = 2$, the assignment $\underline{v_1 = 3}, v_2 = 2$ is in D-space because v_1 is committed to a value that is different value than in the initial assignment and v_2 is uncommitted, but $\underline{v_1 = 1}, v_2 = 2$ is not in D-space because v_1 is committed to the same value as in the initial assignment. Each node in D-space corresponds to a unique assignment of variables to values.[1] Despite the redundancy in C-space compared to D-space, there are some intuitive advantages of C-space. Explicitly committing a variable to its initially assigned value allows us to consider that commitment in the domain-specific algorithms for detecting infeasible states and for computing lower bounds on the number of additional changes required to reach a conflict-free state, allowing us to prune more effectively. Combined with a most-constrained variable ordering heuristic and some heuristic for value ordering, this allows us to identify jobs/requests that are highly constrained and hard to move, and place them early on in the search tree.

2.1 Pruning Infeasible Nodes

A search node N is *infeasible* if there exists no descendant of N that is a conflict-free state. Infeasible nodes can be pruned. The *wasted space* of a bin B is amount of space in the bin that can not be occupied by any uncommitted item currently not assigned to B without making B oversubscribed. For example, suppose we have a bin $B = (\underline{7})$ with capacity 10, and three remaining uncommitted items 7,4, and 2 (which are currently in other bins). The wasted space of B is 1, because the minimal amount of unused space that can be in B is 1, after moving and committing the 2 to B. The wasted space of a bin assignment is the sum of the wasted space of each of the individual bins.

Let $W_{UB} = \sum_{j=1}^{m} c_j - \sum_{i=1}^{n} w_i$, the difference between total bin capacity and total item weights, be an upper bound on the total amount of wasted space allowed. A node in the search tree is infeasible if a lower bound on the wasted space exceeds W_{UB}. Such a lower bound is obtained by summing the lower bound on the wasted space of each single bin, $\sum_{j=1}^{m} W_{LB}(j)$. For each bin j, $W_{LB}(j)$ is computed by find the packing of bin j with minimal wasted space, using all uncommitted items (this is a relaxation because we allow uncommitted items to be used by more than one bin). This is a subset sum problem, which our current implementation solve using a straightforward, branch-and-bound algorithm. The remaining space in the bin after packing the optimal subset-sum packing is a lower bound on the actual wasted space of the bin.

[1] Symmetric nodes are pruned in both C-space and D-space.

	Commitment Space (C-space)						Difference Space (D-space)					
	DFBNB			IDA*			DFBNB			IDA*		
(# bins)	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes
4	0	0.007	1230	0	0.002	149	0	28.516	3455195	0	0.009	839
5	0	0.083	14365	0	0.007	1082	0	32.832	3608220	0	0.112	9103.3
6	0	1.091	188503	0	0.044	5072	**7**	29.868	37833329	0	0.897	63796
8	0	51.011	8580060	0	1.132	110905	**16**	141.160	8569661	**2**	23.196	1326308
10	**20**	n/a	n/a	0	8.010	690901	**20**	n/a	n/a	**11**	51.997	2448719
12	**20**	n/a	n/a	0	58.731	4911799	**20**	n/a	n/a	**14**	95.74	4446458
15	**20**	n/a	n/a	**7**	140.660	10848252	**20**	n/a	n/a	**20**	n/a	n/a

Table 1. Min-Cost Load Rebalancing Problem: Depth-first branch-and-bound (DFBNB) and IDA* in C-space and D-space. The *fail* column indicates # of instances (out of 20) that were not solved within the time limit (300 seconds/instance). The *time* and *nodes* columns show average time spent (seconds on 2.4GHz Intel Core2) and nodes generated on the successful runs, excluding the failed runs.

2.2 Lower Bounds

We use the following lower bound for DFBNB and IDA* (the admissible heuristic h for IDA*). A bin is *oversubscribed* if the sum of the weights of the items assigned to the bin exceeds its capacity. An oversubscription-based lower bound LB_O is computed as follows: For each oversubscribed bin B, sort the uncommitted items assigned to the B in non-decreasing order of weight, and count the number of items that must be removed from B in this order until the bin occupancy no longer exceeds capacity. For example, given the bin assignment $\{(5,6)(4,3)(\underline{10},1,2)\}$ where bin capacity is 10, $LB_O = 3$. This is because either the 5 or 6 must move from the first bin, and the 1 and 2 must move from the third bin (although the 10 is the largest number in the third bin, it is committed so it is not considered for movement by the LB_O computation). Another lower bound, LB_U , is based on the bins that are undersubscribed. If any bin has more free space than W_{UB}, then some of the remaining uncommitted items must move into the bin to reduce the wasted space. A valid lower bound is to add the largest remaining uncommitted items until the free space no longer exceeds W_{UB}. Since both LB_O and LB_U are inexpensive to compute, we use a combined lower bound, $LB_{OU} = max(LB_0, LB_U)$. We can not add LB_0 and LB_U because that may result in the double-counting of the potential required moves.

Note that these bounds, as well as the previously described method of pruning infeasible nodes, are all applicable to to search in both C-space and D-space.

3 Experimental Results

We generated a set of solvable benchmarks as follows. m empty bins were initialized with capacity 100. For each bin b_j, items were randomly generated in the range [10,30] and assigned to b_j until the remaining space r was under 10. At

that point, the slack in the bin was reduced by adding one random 'filler' item such that the remaining space in b_j was between 0 and 2. By minimizing the slack (free space) the difficulty of the instance is increased. Then all the bins were emptied, and the items were combined into a single list, which was shuffled and reassigned to the bins in a round-robin manner. This generation process ensures that the instance has a feasible solution. Because of this generation method, the number of items in each instance varies, but is approximately $5m$. We tested each of the four search algorithm configurations on 20 random instances with m varying from 4 to 15, with a time limit of 300 seconds per instance on a 2.4GHz Intel Core2 processor. Results are shown in Table 1. The *fail* column indicates the number of instances (out of 20) that were not solved within the time limit (300 seconds/instance). The *time* and *nodes* columns show average time spent and nodes generated on the successful runs, excluding the failed runs.

As shown in Table 1, IDA* in C-space significantly outperformed the other three algorithms. Both of the C-space algorithms significantly outperformed the D-space search algorithms, and in both search spaces, IDA* outperformed DF-BNB. Future work will present more detailed results, including the effect of various variable and value orderings, as well as additional bounds.

References

1. G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. In *Proc. 15th ACM Symp. on parallel algorithms and architectures*, pages 258–265, 2003.
2. G. Belov and G. Scheithauer. A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. *European Journal of Operational Research*, 171:85–106, 2006.
3. H. El-Sakkout and M. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5:359–388, 2000.
4. A. Fukunaga and R. Korf. Bin completion algorithms for multicontainer packing, knapsack, and covering problems. *Journal of AI Research*, 28(393-429), 2007.
5. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
6. W. Harvey and M. Ginsberg. Limited discrepancy search. In *Proc. IJCAI*, pp. 607–615, 1995.
7. R. Korf. Improved limited discrepancy search. In *Proc. AAAI*, pp. 286–291, 1996.
8. R.E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
9. S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, 1990.
10. Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.
11. Y.P. Ran, N. Roos, and H.J. van den Herik. Approaches to find a near-minimal change solution for dynamic CSPs. In *Proc. CP-AI-OR*, pages 378–387, 2002.
12. G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proc. AAAI*, pages 307–312, Seattle, Washington, 1994.