# A Primal Only Heuristic Depth First Branch and Bound Approach to the Generalized Assignment Problem

Arjun Lakshmipathy

***Abstract*— We propose an optimized approach to solve the primal Generalized Assignment Problem via a Heuristic Depth First Branch and Bound Search using a combination of variable and node ordering, heuristic computation and pruning, and lower bound pre computation. We focus on solving the primal problem exactly and find the largest problem that can be solved optimally using this technique. We then compare the performance of our algorithm against some results of known state of the art techniques that instead solve dual-feasibility problems.**

## I. INTRODUCTION AND MOTIVATION

The generalized assignment problem, noted as GAP for short, consists of trying to optimize the assignment of a set of n items to a set of m bins, in which n is strictly greater than m. Every item consists of both a cost (dependent on its assigned bin) and a list of "rewards" which depend on the bin it ultimately gets assigned to. Each item can only be assigned to a single bin, though each bin can be assigned multiple items. Furthermore, a bins remaining capacity must either exceed or be equal to a particular items cost in order to be able to take that item. The problem therefore consists of trying to find an assignment which results in either a minimization (in which "rewards" are viewed negatively) or maximization (in which "rewards" are viewed positively) of the sum of the reward values gained from assigning all or a subset of the items to each bin. In the context of this work, we narrow our analysis to integer-only problems of non-negative values. A reduction of a non-integer problem to an integer problem can be made by simply multiplying all the values by a multiple which converts all fractional terms to whole numbers, and therefore the approach can be designed to work without loss of generality. We are interested in analyzing maximization problems in this work, and therefore a mathematical formalization of the problem is as follows:

$$maximize \sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij} x_{ij} \tag{1}$$

$$s.t. \sum_{j=1}^{n} c_{ij} x_{ij} \leq b_i, \qquad i = 1,..,m \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1,...,n \tag{3}$$

$$x_{ij} \in \{0,1\}, \qquad i = 1,...,m, \ j = 1,...,n \tag{4}$$

In the above equations, r represents the "reward" of an item, c represents the items cost, b represents a bins capacity, and x represents the status of the assignment of a particular itme to a particular bin. Moving forward, we refer to equation (1) as the primal equation, consistent with optimization terminology. The GAP has remained a consistent problem of interest for several decades across the fields of operations research, mathematics, and computer science, with the majority of need stemming from applications to operations and resource allocation. Some concrete examples of problems in which a solution to the GAP would be desirable would be in assigning ads to windows, processes to processors, workers to jobs, and a variety of other scenarios that span multiple industries. The problem has especially garnered a fair amount of interest in the research community due to its notoriety of being both NP-hard and even APX-hard - in other words, there exists no known polynomial or even pseudo polynomial algorithm to either exactly or even approximately solve this problem.

## II. RELATED WORK

Ross and Soland[1] are largely credited with presenting the original problem and a subsequent algorithm to solve it using Lagrangian relaxation of constraint (1) (albeit for a minimization problem). Martello and Toth[14] analyzed and subsequently proved that the difference between a minimization and maximization primal problem are largely negligible, and thus techniques could extend along either route. It is worth noting that because this problem is both NP-hard and APX-hard, there are several publications that focus

on good approximation methods that usually find an optimal solution, including the hybrid approach presented by Nauss[2], numerical subgradient descent technique proposed by Karabaka et. al[5], the branch and price + column enumeration method presented by Savelsbergh[3], the tabu heuristic search technique presented by Diaz et. al[8], and even a genetic algorithm approach proposed by Chu and Beasley[10].

In the scope of this work, however, we focus exclusively on methods which are guaranteed to yield optimal solutions, and therefore cannot truly employ some of these techniques. In this regard, we refocus our attention on the technique employed by Ross and Soland, which a considerable number of researchers have attempted to optimize. The technique, as described in the original work but perhaps more succinctly in the survey paper of Cattyrse et. al[12] is as follows:

1) Drop constraint (2) on the primal problem. The resulting problem reduces to one whose optimal solution can be easily achieved by assigning each item to its highest value yielding bin.
2) Starting with this infeasible solution, systematically search for a feasible solution. This can be achieved with a Depth First Branch and Bound search. When an item is moved away from its optimal assignment bin, the move incurs a penalty, and thus the new objective is to optimize the problem which sums together the current assignment and the penalty cost.
3) When a solution is found which satisfies the original bin constraints, use that as the global optimum for future pruning. Exhaustively perform the search until the minimum (or maximum in our case) of all the feasible solutions is found.

Researchers later found that, empirically, dropping constraint (3) instead, which results in a problem whose optimal solution can be found by computing m binary knapsack problems, yields a more promising initial infeasible outcome and requires less searching on average. Nonetheless, this approach is interesting since it essentially converts the problem from an intuitive heuristic search into a dual (referred throughout the remainder of the paper) constraint satisfaction problem, analogous to (but not exactly like) heuristic repair. The normal shortcoming of heuristic repair - endlessly running if no solution exists - is actually mitigated in the context of this problem as a solution is guaranteed to exist. On top of this, Haddadi and Ouzia[4] also introduced a lower and upper bound heuristic. Finally, the latest works include those of Avella et. al[9], which employs a deterministic minimum cut scheme, and Post et. al[6], which employs variable fixing and a systematic search of feasible solutions through increasing upper bounds (in the context of a minimization problem) until one is found, at which point it is determined as optimal.

What's interesting in this survey findings is that not one of these authors attempted to tackle the primal problem, which this author believes can be solved directly also using a Depth First Branch and Bound search. In this work, therefore, we aim to explore the possibility of solving the primal problem itself optimally using a combination of static variable ordering, initial lower bound computation, node ordering, heuristic computation, and pruning scheme. We then observe the performance compared to the dual constraint satisfaction approach. In other words, we compare the performance of the best primal heuristic search technique we can come up with the approaches proposed in the literature. The advantage of solving the primal directly is that only the set of feasible solutions will be explored in the first place, and the algorithm can be terminated at any time to yield either a reasonably good, if not already-discovered optimal, solution. Furthermore, we have the guarantee that it will always find an optimal solution when the entire search terminates.

## III. PROBLEM SPACE

Starting from the root node, every subsequent node represents a state of the search space. Each of these states is comprised of two components: (i) a map containing a list of bins and the items assigned to each bin (ii) each bins remaining capacity and (iii) a list of remaining unassigned items. Additional components of items will be elaborated on in following sections for use in static value and heuristic computations. An edge in this case represents a move in which item i is assigned to bin j, the consequence of which is an update of the current running utility value (i.e. the primal total) with the value of assigning item i to bin j and a reduction of bin js capacity by the cost of assigning item i. This naturally results in a bounded branching factor that is equivalent to exactly the number of bins, plus the addition of one (which represents not assigning an item to any bin).

There are multiple goal states in this case, and potentially several which could yield an optimal solution path. In the case of there being multiple optimal solutions, we are satisfied with only finding one of

them - it is therefore ok to prune off a potential second optimal solution even if its equal to one we have already found. A goal state, in this case, is a node in the search tree which satisfies at least one of the following criteria:

1) There are no more items to assign
2) The lowest cost item cannot fit into even the bin of highest remaining capacity
3) All of its children have been pruned either for resulting in illegal states or by the heuristic computation

In addition, there is a globally running optimal value which is compared against a goal nodes final resulting cumulative reward value prior to continuation of the search. If the cumulative reward value of the goal state exceeds the global optimum, its resulting optimal value and assignment become the new global optimum. An optimal goal state therefore is one which, at the end of the search, remains the global optimum value.

An initial ordering of the items is selected at the start of the search and maintained throughout. Therefore each item i will only be assigned and depth i of the search tree. Constructing the problem and assigning items sequentially in this manner completely removes all potential duplicate nodes from the search space and thus allows for the problem space to be entirely represented as a finite tree. We can therefore employ a Depth First Branch and Bound approach and have a strong guarantee of termination while gaining the added benefit of only linear space complexity. For this reason, DFBnB was the natural choice of search through this problem space. The remainder of this paper will focus on detailing the specific of the search itself.

## IV. ALGORITHM CONSTRUCTION

The heuristic Depth First Branch and Bound algorithm was divided into four parts, each of which is explained in greater detail below.

### A. Variable and Node Ordering

*1) Static Variable Ordering:* Four possible static ordering schemes were devised, which were based on: (i) the highest value an item can provide when assigned to its optimal bin (ii) the highest ratio an item can yield (computed by dividing its highest value divided by the cost) (iii) the highest frequency of usage when solving an m binary knapsack problem prior to performing the search and (iv) a hybrid model which used (iii) as a primary criteria and (i) as a secondary criteria. Surprisingly, out of these four schemes (i) yielded the highest amount of pruning both in manually generated and scalability oriented test instances, and was therefore selected as the static variable ordering sorting criteria of choice.

*2) Dynamic Node Ordering:* The children of any node were also ordered in both increasing actual utility (g) values and increasing estimated composite utility (f = g + h) and analyzed on problems of both manually generated and randomly generated instances of varying size. Again, surprisingly, ordering based on g values alone consistently outperformed sorting based on f values - this difference became especially pronounced in larger instances. Therefore, while not as intuitive as sorting by f values, this was the sort criteria chosen.

### B. Heuristic Determination

An admissible heuristic was derived by beginning with a STRIPS formulation of the problem, which consisted of the following preconditions:

- ASSIGNED(X,B) - Returns true if item X has been assigned to bin B. B can also be empty, which is represented by U
- HAS_CAPACITY(B,XC) - Returns true if bin B has enough capacity to accept an item of cost XC and false otherwise

A valid move therefore is as follows: MOVE(X,B) would result in ASSIGNED(X,B) if and only if, prior to the move being executed, the preconditions ASSIGNED(X,U) (item X is not assigned to a bin currently) and HAS_CAPACITY(B,XC) (XC is the cost of item X in this case) both are true. We then derive an admissible heuristic by relaxing one of these preconditions.

First we examine the scenario of dropping HAS_CAPACITY. The equivalent problem generated consists of all of the bins essentially having infinite capacity, and therefore has an optimal solution which can be achieved by simply assigning each item to its highest value bin. This can be solved by a simple linear pass through the data and is therefore a reasonably cheap heuristic to compute.

In contrast, dropping the ASSIGNED precondition results in an equivalent problem that allows replacement of items (in other words items can be used more than once). This would mean that the solution to this problem would involve performing a knapsack evaluation on each of the bins with all the remaining items present, which would result in pseudo-polynomial time complexity of $O(mnW_m)$, where m represents the number of windows, n the number of remaining items, and $W_m$ representing the capacity of a particular bin m.

This analysis results in the exact relaxations mentioned by dropping the Lagrangian constraints to form dual problems mentioned in the literature, which is useful to note as, effectively, these relaxations never underestimate the true cost to the goal state. In this regard the dual problems essentially also serve as admissible heuristics, which we employ in the approach to solving the primal directly.

A third but perhaps non-intuitive heuristic can be derived by also dropping constraint (4) in the original problem going off this observation. This allows for a problem in which items can be "partially" assigned to bins, and the solution is again an m knapsacks problem (but not a binary m knapsacks problem). The problem therefore becomes finding optimal weights to assign to each item such that its utility is distributed optimally among the various bins, and an examination is conducted by Benders et. al[15]. However, we consider an analysis of this heuristic outside the scope of this work and therefore merely mention it for the sake of completeness. That being said, it is a very viable route to examine for future work.

Several papers have shown that, in practice, relaxing ASSIGNED (equivalently) results in approximations that are much closer to the actual solutions than those determined by relaxing HAS_CAPACITY and can therefore, in a reasonable number of scenarios, be expected to prune more nodes; however, the difference in computational complexity is quite substantial in the two approaches. Furthermore, this relaxation is NOT GUARANTEED to do strictly better than the first heuristic since it still is ignoring a precondition - it will simply give a different approximation that is guaranteed to be admissible. That being said, we note an inherent problem in the effectiveness of the first scenario created by variable ordering. Once a variable is selected for assignment to a particular bin, its children will all have the same computed value according to the first heuristic since remaining capacities of the bins are ignored - this of course is not very useful when it comes to effective pruning or path selection as it does not at all consider the fact of a bin having been used by assignment of the variable in the parent node. To remedy this, we modify this heuristic by instead computing the max value an item can impart when assigned to a bin that has sufficient capacity to accept it. To do this we can simply precompute an ordered list of bins in decreasing order of value imparted if the item was assigned to a particular bin; therefore, we simply traverse down the list until a bin which still has enough capacity to support the item is found (and if not, the value would be 0). At the cost of a little extra complexity and precomputing time, this would result in a significantly more useful heuristic as it actually takes the current state into account and still can be guaranteed to never to underestimate the optimal solution.

But the more interesting point to note here is that because BOTH of these heuristics are admissible and provide two fairly different viable solution estimates, a "third" heuristic can be formed by simply combining (not summing) the information from both computations together. This scheme will be further elaborated upon in the proceeding section.

### C. Pruning Scheme

The general rule of DFBnB search applies in the context of this algorithm - if the computed sum of a generated child node, which is a combined total of the static value of an assignment in a particular state and the heuristic value of that state, does not strictly exceed the currently known global optimum value, then the child node is pruned. Additionally, as was the case even with the brute force search, any child states that are illegal (bin capacity is exceeded) will also be pruned.

The more interesting scheme comes into play when combining the two heuristic schemes together. Because the returned computation from both heuristics is guaranteed to never underestimate the true cost to the goal, the observation here is that this results in a scenario where these values are either both equal or not. In the case of inequality, we actually can simply choose the lower of the values to apply to the situation. In other words, in order for a node to not get pruned, BOTH its computed sums must exceed the global optimum value discovered so far in order for the node to be considered viable for expansion. The resulting pruning scheme will prune at least the same number of nodes as either of the individual pruning schemes alone, but at the same time makes use of the information gained by doing both computations simultaneously. Therefore, on average, this dual heuristic evaluation scheme can be expected to do at least slightly better than using either individual heuristic alone at cost of introducing slightly more run time complexity. However, this complexity is largely dominated by the m knapsack heuristic and therefore would take, at least in big O asymptotic terms, the same time to evaluate as the m knapsack heuristic alone. The proof of this pruning scheme never pruning off nodes on the optimal path is rather trivial, and is therefore omitted from the report for conciseness.

### D. Lower Bound Pre-computation

DFBnB cannot perform any pruning until it reaches a goal node; however, it is perfectly reasonable to pre-compute a lower bound prior to beginning the search. Depending on the specifics of the actual solution, it may be possible that the pre computed lower bound may even generate an optimal solution and therefore allow for pruning off all the children of the root immediately. That being said, the pre computed solution, in the case of a maximization primal problem, must NOT overestimate the actual optimal solution payout. Furthermore, such a solution should be computed either polynomial or at worst pseudo-polynomial in complexity in order to be evaluated in a small amount of time and therefore not dominate the heuristic search itself.

In this regard, two possible values come to mind: the maximal binary knapsack value yielded over all the bins, and the value yielded from solving the singular assignment problem. The former can be computed in pseudo-polynomial time by running knapsack over all the bins and taking the max value as the global optimum, while the latter can be solved in polynomial time using the Hungarian algorithm for variable assignment[16]. Both of these approaches solve essentially simplified versions of the problem, and thus are guaranteed to never overestimate the true value of the optimal solution. Combined, the knapsack solution will allow the algorithm to trivially find solutions in which all, or at least nearly all, items are assigned to a single bin, while the assignment solution will trivially solve all problems in which the solution involves assigning a single item to every bin. Note here that the assignment algorithm expects an equal number of items and bins, and therefore we pre sort and select the top m items based on the static variable ordering mentioned previously.

Obviously the solution to the actual primal problem will most likely deviate substantially from either of these two extremes; however, there is no reason NOT to do these sub problems first since they do not sacrifice solution quality nor amount of pruning. Therefore, the additional overhead incurred by this pre computation is certainly justified in comparison to the amount of pruning it could result in based on the initial lower bound either can present. The knapsack solution is computed as time-optimally as possible using the approach in[7], and the Hungarian algorithm equally as such using approaches widely available on the web.

## V. EXPERIMENTS AND RESULTS

The algorithm was first implemented in Python and then ported to a C implementation in order capitalize on faster computation time. Experimentation involved two classes of tests: small instances to check for correctness and optimality of solutions, and larger tests intended to test for scalability. 7 small instances were optimally computed by hand to determine both the optimal assignment and the resulting utility value for said assignment. Due to the author's personal computer having pitiful specs and crashing for larger problem instances due to CPU overuse, all values reported in the following tables are those output by running the algorithm on a Google Cloud partition sporting multiple Intel Xeon 2.2 GHz processors (in order to run several searches simultaneously). All run times are reported in seconds.

A brute force search was first implemented and compared to the optimal Python and later C implementations - the results of which can be seen in Table I, Table II, and Table III. As expected, there is a monumental difference in both run times and nodes expanded between the brute force and the more optimal searches. As can be seen in Table I, the brute force search struggles even on small instances and therefore was not run on larger problem instances.

The reader may notice an oddity in the tables stemming from the difference in the number of nodes expanded by the C and Python implementations. Intuitively the expectation would be that both programs should expand the exact same number of nodes; however, the difference created here, after many hours of testing and investigation, stems from the difference in tie breaking of the sorting algorithms between the two languages. The C standard library quick sort was stabilized to adhere to the sorting scheme presented in the Python documentation to yield consistent results for tie breaks based on the original position of the values; however, the Python library sort continuously failed to adhere to a strict secondary initial position sort criteria. As such, despite using the documented sort determination scheme, the exact sort criteria proved to yield different results from the Python sort for different test instances (in other words while it held true for some, it did not hold true for others). This of course would not at all affect the finding of an optimal solution; however, as evident from the tables, the difference in the number of nodes expanded can be quite substantial. In this regard, the difference in run time of these algorithms is significantly impacted by

the tie breaking scheme of the sort criteria, which was a fairly unexpected result.

M7 was crafted as a special test case in that it was heavily biased in favor of a particular bin. As can be seen from the generated nodes and run time, this solution, despite involving a large number of variables, was solved instantly by the precomputed Knapsack lower bound exceeding the projected value of any of the child nodes. A similar test case was constructed for the purpose of demonstrating the utility of the Hungarian Algorithm lower bound, but was neglected on account of repetition.

From observing the Python and C tables for the manual test cases, it is fairly evident that a C implementation is substantially faster on average; however, the "unknown" tie breaking sort of the Python implementation often saves considerably on the number of nodes expanded. Further work needs to be conducted in order to fully align the sorts together to make the algorithms even more easy to directly compare (though time limitations of the quarter prevent such work for the time being). That being said, both algorithms solved all the manual instances optimally and in reasonable time amounts.

The scalability instances were taken from the JE Beasley OR (http://people.brunel.ac.uk/ mastjjb/jeb/orlib/gapinfo.html). Some of the instances available in this library extend to thousands of variables; however, it was expected that the implementations proposed in this paper would not make it that far. Table IV, Table V, and Table VI show the results of runs again of both the Python and C implementations. The Python instance is able to solve some instances of nearly 200 variables, though others timed out on account of running into the 15000 second time limit. What is particularly interesting to note is the substantial variance in run times and node expansions even on problems of the same size - it turns out that the nature of the solution substantially alters the performance of the algorithm with regards to pruning.

Unfortunately, due to only having a final scalability instance C version ready around 12 am the night before this final paper was due (it took a considerable amount of reorganizing of code - and a TON of segmentation faults and valgrind debugging of memory leaks - to have it adhere to the nuances in the scalability instances, which were borrowed, as opposed to the manually generated home made instances), this implementation was only able to be successfully run on instances up to 100 variables. In the interest of evaluating more instances for the Python table, the author unfortunately chose to sacrifice some development time for the C implementation. This is an unfortunate result of time constraints as the results looked really promising it is likely that the C implementation could solve at least this many instances, if not substantially more than the Python implementation given its current trajectory. We therefore need to hypothesize some limits of the C implementation based only on the current data collected.

By roughly extrapolating the results of the C implementation on the scalable instances and its performance difference on the manual instances, the author roughly estimates that this implementation will be able to solve instances of around 250 (or possibly more, depending on the instance) or so variables. This is pretty reasonable with regards to solving the primal problem directly, especially given the exponential nature of the problem. The initial expectation was that the algorithm would begin to seriously struggle at around 130 variable mark; however, even the Python implementation is able to expand to considerably more variables, which is substantial considering the tree is growing exponentially.

## VI. CONCLUSIONS

The highly optimized algorithm presented in this work to solve the primal problem directly was able to optimally solve instances of around 150 to 200 variables (and supposedly around 250 with given more time with the C implementation), which is an enormous improvement over the pitiful performance of a brute force search. Furthermore, as a result of the pre computation steps, problem instances with highly biased optimal solutions could be solved very quickly and thus requires very little searching at all. However, it is somewhat limited when it comes to expanding to even larger problems which are more evenly distributed item allocations. The works presented by some of the literature in this domain are able to solve several hundreds of variables of variables optimally, and even upwards of several thousands in the case of approximate algorithms (see tables from the reference papers). This is not to say that the approach presented in the paper is by any means bad as it still does a very good job in pruning out lots of unnecessary nodes compared to a brute force search. On top of that, because it only searches through feasible solutions, it can be terminated at any time to yield a perhaps sub optimal, but fully feasible solution in comparison to the dual problem techniques. However, it is still somewhat limited in the sizes of average problems that it can solve optimally.

TABLE I: Brute Force - Manual Test Cases

| Test Case | Number of Bins | Number of Items | Total Variables | Generated Nodes | Run Time |
|---|---|---|---|---|---|
| M1 | 3 | 6 | 18 | 1035 | 0.130805969238 |
| M2 | 3 | 6 | 18 | 946 | 0.144660949707 |
| M3 | 4 | 6 | 24 | 1195 | 0.149334907532 |
| M4 | 6 | 8 | 48 | 44955 | 18.9128739834 |
| M5 | 2 | 4 | 8 | 25 | 0.00245714187622 |
| M6 | 8 | 10 | 80 | 3126355 | 3741.16774702 |
| M7 | 10 | 20 | 200 | TIME | TIME |

TABLE II: Python Implementation - Manual Test Cases

| Test Case | Number of Bins | Number of Items | Total Variables | Generated Nodes | Run Time |
|---|---|---|---|---|---|
| M1 | 3 | 6 | 18 | 27 | 0.00584387779236 |
| M2 | 3 | 6 | 18 | 19 | 0.00441288948059 |
| M3 | 4 | 6 | 24 | 16 | 0.00356793403625 |
| M4 | 6 | 8 | 48 | 43 | 0.0311119556427 |
| M5 | 2 | 4 | 8 | 6 | 0.000906944274902 |
| M6 | 8 | 10 | 80 | 109 | 0.427762985229 |
| M7 | 10 | 20 | 200 | 1 | 0.307541847229 |

TABLE III: C Implementation - Manual Test Cases

| Test Case | Number of Bins | Number of Items | Total Variables | Generated Nodes | Run Time |
|---|---|---|---|---|---|
| M1 | 3 | 6 | 18 | 26 | 0.001348 |
| M2 | 3 | 6 | 18 | 39 | 0.001920 |
| M3 | 4 | 6 | 24 | 37 | 0.001762 |
| M4 | 6 | 8 | 48 | 64 | 0.010437 |
| M5 | 2 | 4 | 8 | 10 | 0.000263 |
| M6 | 8 | 10 | 80 | 194 | 0.069697 |
| M7 | 10 | 20 | 200 | 1 | 0.001696 |

We therefore are lead to conclude that dual optimization techniques are more promising than solving the primal problem directly when it comes to scalability, even if the primal approach is highly optimized. The work presented in this paper therefore highlights the power of dual constraint satisfiability approaches by comparing them against a primal only approach, and therefore suggests that further research should be directed toward improving search techniques along those lines for solving even larger problem instances. That being said, the focus of this work in coming up with, testing, and comparing a pure primal DFBnB algorithm against the state of the art was carried out reasonably successfully.

## VII. FURTHER WORK

Due to a combination of completing an actual research paper this quarter and being limited in man power and compute power, there remains a substantial amount of work left to do in order to further this work. First, more time must be allocated to allow the C implementation to run on the scalability instances in order to fill out more table entries - at least as many as that of the Python implementation. The principal objective here would be to observe whether this implementation is able to scale to 250 variables or more (and if so how much more) in order to see if it approaches the results presented by dual optimizations. If it indeed can approach those numbers, then there is some value

TABLE IV: Python Implementation - OR Test Cases

| Test Case | Number of Bins | Number of Items | Total Variables | Generated Nodes | Run Time |
|---|---|---|---|---|---|
| GAP1-1 | 5 | 15 | 75 | 26962 | 24.6288700104 |
| GAP1-2 | 5 | 15 | 75 | 1360 | 2.21799778938 |
| GAP1-3 | 5 | 15 | 75 | 853 | 1.36122703552 |
| GAP1-4 | 5 | 15 | 75 | 2164 | 3.44639492035 |
| GAP1-5 | 5 | 15 | 75 | 18356 | 20.3537549973 |
| GAP2-1 | 5 | 20 | 100 | 6799 | 20.5834131241 |
| GAP2-2 | 5 | 20 | 100 | 20644 | 61.9456591606 |
| GAP2-3 | 5 | 20 | 100 | 7156 | 24.2434952259 |
| GAP2-4 | 5 | 20 | 100 | 16170 | 43.8173260689 |
| GAP2-5 | 5 | 20 | 100 | 1891 | 5.01964497566 |
| GAP3-1 | 5 | 25 | 125 | 44136 | 188.793950081 |
| GAP3-2 | 5 | 25 | 125 | 1467555 | 5224.17732 |
| GAP3-3 | 5 | 25 | 125 | 39827 | 227.348176003 |
| GAP3-4 | 5 | 25 | 125 | 105682 | 453.358449936 |
| GAP3-5 | 5 | 25 | 125 | 1548023 | 5505.73040199 |
| GAP4-1 | 5 | 30 | 150 | 515121 | 5185.65801692 |
| GAP4-2 | 5 | 30 | 150 | TIME | TIME |
| GAP4-3 | 5 | 30 | 150 | 265396 | 3201.29728484 |
| GAP4-4 | 5 | 30 | 150 | 1795444 | 13387.257051 |
| GAP4-5 | 5 | 30 | 150 | 503498 | 4052.89386511 |

TABLE V: Python Implementation - OR Test Cases (cont.)

| Test Case | Number of Bins | Number of Items | Total Variables | Generated Nodes | Run Time |
|---|---|---|---|---|---|
| GAP5-1 | 8 | 24 | 192 | 210972 | 2579.29847002 |
| GAP5-2 | 8 | 24 | 192 | 1708992 | 12721.0854518 |
| GAP5-3 | 8 | 24 | 192 | 98564 | 1650.79488397 |
| GAP5-4 | 8 | 24 | 192 | 53810 | 659.104093075 |
| GAP5-5 | 8 | 24 | 192 | TIME | TIME |

TABLE VI: C Implementation - OR Test Cases

| Test Case | Number of Bins | Number of Items | Total Variables | Generated Nodes | Run Time |
|---|---|---|---|---|---|
| GAP1-1 | 5 | 15 | 75 | 1106 | 0.554815 |
| GAP1-2 | 5 | 15 | 75 | 4287 | 2.516490 |
| GAP1-3 | 5 | 15 | 75 | 598 | 0.272636 |
| GAP1-4 | 5 | 15 | 75 | 10877 | 3.764925 |
| GAP1-5 | 5 | 15 | 75 | 5621 | 1.887374 |
| GAP2-1 | 5 | 20 | 100 | 17415 | 8.318447 |
| GAP2-2 | 5 | 20 | 100 | 31255 | 15.197031 |
| GAP2-3 | 5 | 20 | 100 | 18041 | 10.138848 |
| GAP2-4 | 5 | 20 | 100 | 32798 | 19.750667 |
| GAP2-5 | 5 | 20 | 100 | 13416 | 6.429487 |

in even pure primal approaches that further research could help improve rather than optimizations purely of the dual problem. Should this not prove to be the case, however, then the work can be refocused at using techniques gained from this work and applying them to further optimize dual problem approaches.

It is also worth noting that, in this work, we focused our attention purely on serial approaches; however, as with most tree search algorithms, the search can be easily parallelized and gain considerable performance improvement. A parallelization of the dual problem was actually explored by Asahiro et. al[11], but no work has been done so far with regards to a direct primal approach. Therefore this would also be a considerable avenue of interest for further work as well.

In any case, a considerable amount of additional time can certainly yield further insights into solving the GAP optimally, which still remains a considerable problem of interest in the academic community across a variety of disciplines.

## REFERENCES


[1] G.T. Ross and R.M. Soland. A Branch and bound algorithm for the generalized assignment problem. *Mathematical programming*, 8(1):91-1-3, 1975

[2] R.M. Nauss. Solving the generalized assignment problem: An optimizing and heuristic approach. *INFORMS Journal on Computing*, 15(3):249-266, 2003

[3] M. Savelsbergh. A branch-and-price algorithm for the generalized assignment problem. *Operations Research*, 45(6):831-841, 1997

[4] S. Haddadi and H. Ouzia. Effective algorithm and heuristic for the generalized assignment problem. *European Journal of Operational Research*, 153(1):184-190, 2004

[5] N. Karabakal, J.C. Bean, and J.R. Lohmann. A steepest descent multiplier adjustment method for the generalized assignment problem. Technical report, University of Michigan 1992.

[6] M. Posta., J.A. Ferland, P. Michelon: An exact method with variable fixing for solving the generalized assignment problem. Comput. Optim. Appl. 52(3), 629?644 (2012)

[7] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research* 45(5):758-767,1997

[8] J.A. Diaz and E. Fernandez. A tabu search heuristic for the generalized assignment problem. *European Journal of Operational Research*, 132(1):22-38, 2001.

[9] P. Avella, M. Boccia, and I. Vasilyev. A computational study of exact knapsack separation for the generalized assignment problem. *Computational Optimization and Applications*, 45(3):543-555, 2010.

[10] P.C. Chu and J.E. Beasley. A genetic algorithm for the generalized assignment problem. *Computers and Operations Research*, 24:17-23, 1997.

[11] Y. Asahiro, M. Ishibashi, and M. Yamashita. Independent and cooperative parallel search methods for the generalized assignment problem. *Optimization Methods and Software*, 18(2):129-141, 2003.

[12] D. Cattyrse, M. Salomon, L. Van Wassenhove. 1992. A survey of algorithms for the generalized assignment problem.

[13] Fisher, M.L., Jaikumar, R. and Van Wassenhove, L. (1986), "A multiplier adjustment method for the generalized assignment problem", *Management Science* 32/9, 1095-1103.

[14] Martello, S. and Toth, P. (1981), "An algorithm for the generalized assignment problem", in: J.P. Brans (ed.), *Operational Research* '81, North-Holland, Amsterdam, 589-603.

[15] Benders, J.F. and van Nunen, J.A.E.E. (1983), "A property of assignment type mixed integer linear programming problems", Operations Research Letters 2/2, 47-52.

[16] Harold W. Kuhn, "The Hungarian Method for the assignment problem", Naval Research Logistics Quarterly, 2: 83?97, 1955.