Implicit enumeration with dual bounds from approximation algorithms

Nelson Frew

Introduction

Within Mixed-Integer Programming (MIP), [7] we are concerned with solving optimisation problems where at least one of the variables is restricted to be integer. The state of the art method for doing this is the Branch-and-Bound algorithm, to which the reader is referred to [4] for details. The manner in which dual bounds are obtained at within the algorithm is pivotal to its efficiency; however, the general method of relaxing variables to a continuous domain and finding a bound from this solution can be arbitrarily poor. In this report, we present a method of computing dual bounds by using the guarantees provided by approximation algorithms (AAs). We show the current form of the proof of concept for this method, and outline the next steps to be undertaken following this.

Background

The inception of Dantzig's Simplex method [2] in 1947 created a shift in research from linear programming to discrete optimisation problems, which led to what is now known as Land and Doig's Branch-and-Bound algorithm. Simultanously, this same shifting of research questions gave rise to Computational Complexity Theory [1] which provided the foundations for the field of approximation algorithmss. It was not until Wolsey [6] that the Branch-and-Bound was used in light of the rise of approximation research, where he provided a general analysis technique for related approximation worst-cases with optimal LP relaxation solutions to achieve implicit enumeration. However, as far we know, no further work has been done in this area.

Methodology

To demonstrate the potential of using Branch-and-Bound with approximations, first we will describe the theoretical groundwork for this project, and then describe the implementation details of the applications at this point.

Dynamic programming algorithms for the 0,1 Knapsack

For the scope of the proof of concept, we look at using Branch-and-Bound with approximations in solving the 0,1 Knapsack problem (KP). We describe the process of constructing a *fully polynomial time approximation scheme* (FPTAS) for KP. The strategy for deriving an FPTAS operates on using an existing Dynamic Programming algorithm (DP) with a modified version of the original problem. We briefly examine two ways of achieving this.

The first DP discussed in this section is the same as is introduced by Vazirani [5] on his section about KP. We define KP as follows: given n items with associate weights w_i and values v_i, and weight capacity W,

maximise $\sum_{i=1}^{n} v_i x_i$

subject to $\sum_{i=1}^{n} w_i x_i \leq W$ and $x_i \in \{0, 1\}$

A DP to solve this as described by [5] is as follows: let P be the value of the most valuable object in the item set, and let $A(i,p)$ be the minimal weight of the solution to KP with only the first i items available, where the total value is exactly p. If no such set can exist (that is, the first i items cannot yield a value of p and so cannot have an associated minimal weight), $A(i,p) = \infty$. The DP recurrence can be defined as follows:

$$A(i+1,p) = \begin{cases} \min\{A(i,p), w_{i+i} + A(i,p-v_{i+1})\}, \text{ if } v_{i+1} < p \\ A(i+1,p) = A(i,p) \text{ otherwise} \end{cases} \tag{1}$$

The solution is given, then, by finding $max\{p|A(n,p) \leq W\}$. This provides exact computation in $O(n^2P)$, as shown by [5].

The second DP of interest is shown in [3]. For this, we maintain an array, where each entry $A(j)$ for $j = 1, ..., n$ is a solution pair (w,p). An entry $A(j) = (w,v)$ indicates that there is a solution from the first j items that has weight w and value v. The concept of *domination* is used to establish an algorithm in this case. A pair (w,v) dominates pair (w',v') if $w \leq w'$ and $v \geq v'$.

The pseudocode for the DP is, then:

Algorithm 1: Second Dynamic Programming Algorithm for the 0,1 Knapsack

Data: A set of n items with associated weights and values
Result: The weight and value pair of the optimal solution

```
A(1) ← {(0,0), (w_1, v_1)};
for j ← 2 to n do
    A(j) ← A(j − 1);
    foreach (w', v') ∈ A(j − 1) do
        if w' + w_j ≤ B then
            Add (w' + w_j, v' + v_j) to A(j);
        end
    end
    Remove dominated pairs from A(j);
end
return max_{(w,v)∈A(n)} v
```

Approximation algorithms introduction

The FPTAS for KP involves simplifying the original problem instance slightly by ignoring some amount of the least significant bits on the values, parameterised by an introduced error parameter ϵ. Then, we simply use a provided DP to solve the simplified problem, which provides us with our FPTAS (see [5] [3] for detailed treatments).

Algorithm 2: FPTAS for Knapsack

Data: A KP problem instance, and error parameter ϵ.
Result: The weight and value pair of the optimal solution

```
Given ε, let K = εP/n;
For each object i, define adjusted values v' = ⌊v_i / K⌋;
Run a DP with these adjusted item values;
Return the approximate solution set, S', provided by the DP;
```

A key feature of Approximation Algorithms (AAs) is that they provide a guarantee on the quality of the solution returned. For an FPTAS, the solution is guaranteed to be within a factor of $1 - \epsilon$ of the optimal value (for maximisation problems).

Let $z_{S'}$ be the value of solution set S' obtained from the FPTAS, and z_O be the value of the optimal solution set O. Also, let $z'_{S'}$ be the value of the solution set with adjusted/truncated values for the FPTAS's solution set S', and z'_O be the optimal solution set with similarly adjusted values for all items.

If we scale $z'_{S'}$ and z'_O both up by their truncated factor K, we find that:

$$z_O - nK = z_O - \epsilon P \leq K \cdot z'_O \leq K \cdot z'_{S'} \leq z_{S'}$$

And since $P \leq z_O$, we obtain our bounds:

$$(1 - \epsilon) \cdot z_O \leq z_{S'} \leq z_O$$

Implementation of approximation schemes

The current implementation for the FPTAS is as follows (details relevant to its Branch-and-Bound implementation have been omitted).

```
void FPTAS(double eps,int *profits, int *weights, int *x, int *sol_prime,
          const int n, int capacity, const int z, const int sol_flag,
          const int bounding_method, const char *problem_file, double *K,
          int *profits_prime, const int DP_method, const int *variable_statuses)
{
  /* Find max profit item */
  int P = DP_max_profit(profits, n);

  /* Define K */
  *K = define_K(eps, P, n);

  /* Derive adjusted profits */
  make_profit_primes(profits, profits_prime, *K, n, variable_statuses);

  if(capacity >= 0)
  {
    /* Now with amended profits list "profits_primes," solve the DP */
    if (DP_method == VASIRANI)
      DP(profits_prime, weights, x, sol_prime, n, capacity,z, sol_flag,
        bounding_method, problem_file);

    else if (DP_method == WILLIAMSON_SHMOY)
    {
      /* Make problem item struct array */
      struct problem_item items_prime[n];
      for(int i = 0; i < n; i++)
      {
        items_prime[i].weight = weights[i];
        items_prime[i].profit = profits_prime[i];
      }
      int result = williamson_shmoys_DP(items_prime, capacity, n, sol_prime);
    }
  }
}
```

Use of these algorithms within Branch-and-Bound

While AAs can be used within a Branch-and-Bound to obtain primal bounds on the optimal solution, such an approach does not address the need for dual bounds to facilitate pruning. As stated earlier, the approximations provide us with the bound

$$z_O - nK = z_O - \epsilon P \leq K \cdot z'_O \leq K \cdot z'_{S'} \leq z_{S'} \leq z_O$$
$$\rightarrow (1-\epsilon) \cdot z_O \leq z_{S'} \leq z_O$$

which initially provided us with the *a priori* bound. This information guarantees that our optimal value must be in the range:

$$z_O \in \left[z_{S'}, \frac{z_{S'}}{(1-\epsilon)} \right]$$

However, having found the FPTAS solution set S', we compute the value, $K \cdot z'_{S'}$ which, being both less than $z_{S'}$ and parametered by the quality guarantee $K = \epsilon P$, provides us with tighter bounds:

$$z_O \in \left[z_{S'}, \frac{K \cdot z'_{S'}}{(1-\epsilon)} \right]$$

Where the upper bounds here are able to be used as dual bounds for KP in the Branch-and-Bound.

Algorithm 3: General FPTAS-enabled Branch-and-Bound algorithm pseudocode

Data: A KP problem instance
Result: The optimal solution value and set

```
1  initialisation;
2  while Node queue is not empty do
3  |  Get next node;
4  |  if Root Node then
5  |  |  Initialise Global Lower Bound (GLB) with FPTAS;
6  |  else
7  |  |  Find UB and LB for this node using FPTAS;
8  |  end
9  |  if UB < GLB then
10 |  |  Prune this branch and go to step 2;
11 |  end
12 |  if LB > GLB then
13 |  |  GLB ← LB;
14 |  end
15 |  Let UB_p ← current node's parent's upper bound;
16 |  if UB > UB_p then
17 |  |  UB ← UB_p;
18 |  end
19 |  if UB > GLB then
20 |  |  Choose a variable to branch on;
21 |  |  Generate the nodes from the branched variable, and put them on the node queue;
22 |  end
23 end
24 return GLB
```

The implementation for this is as follows (benchmarking and logging features have been omitted):

```
void branch_and_bound_bin_knapsack(int profits[], int weights[], int x[],
                                   int capacity, int z, int z_out, int sol_out[],
                                   int n, char *problem_file,
                                   int branching_strategy, time_t seed,
                                   int DP_method, int logging_rule,
                                   FILE *logging_stream, double eps,
                                   int *number_of_nodes,
                                   int memory_allocation_limit,
                                   clock_t *start_time, int timeout)
{
  LL_Problem_Queue *node_queue = LL_create_queue();
  Problem_Instance *root_node = define_root_node(n);
  Problem_Instance *current_node;
  int first_iteration = TRUE;
  int global_lower_bound;

  /* While node queue is not empty */
  while((first_iteration) || (node_queue->size >= 1))
  {
    /* Get next node */
    if(first_iteration)
    {
      current_node = root_node;
      current_node->upper_bound = INT_MAX;
      global_lower_bound = find_heuristic_initial_GLB(profits, weights, x, z,
                                                      n, capacity,
                                                      problem_file, DP_method);
      current_node->lower_bound = global_lower_bound;
      first_iteration = FALSE;
    }
    else
    {
      current_node = select_and_dequeue_node(node_queue);
      find_bounds(current_node, profits, weights, x, capacity, n, z,
                  &current_node->lower_bound, &current_node->upper_bound,
                  problem_file, DP_method, logging_rule, logging_stream, eps);
    }
    /* If UB < GLB, we safely prune this branch and continue to loop */
    if((current_node->ID != 0) &&
      ((current_node->upper_bound <= global_lower-bound) ||
       (current_node->upper_bound > curent_node->parent->upper_bound)))
      continue;

    /* If for N's parent's upper bound, UB_p, UB > UB_p, UB = UB_p */
    if (current_node->ID !=0) &&
        current_node->upper_bound > current_node->parent->upper_bound)
      current_node->upper_bound = current_node->parent->upper_bound;

    /* If UB > GLB, we branch on the variable */
    if (current_node->upper_bound > global_lower_bound)
    {
      branching_variable = find_branching_variable(n, z, current_node->variable_statuses,
```

```
                                                    branching_strategy, profits);
      if(branching_variable == -1) // Leaf node
        continue;
      generate_and_enqueue_nodes(current_node, n, branching_variable, node_queue,
                                 &count, logging stream, logging_rule,
                                 &node_limit_flag);
    }
  }
  *z_out = global_lower_bound;
}
```

Benchmark results

Below are preliminary benchmarks with the following characteristcs:Memory allocation is constrained no more than 10000000 bytes, and run time is constrained to 120 seconds. The problems addressed are those with n=50, 100, and 200, instance types 1 (Uncorrelated), 2 (Weakly correlated), and 3 (Strongly correlated), and coefficients of size 1000, 10000, all using Williamson Shmoy's DP strategy. The branching strategy is informed by the amount a variable will be truncated in the FPTAS.

instance	n	coeffcients	problem #	runtime	memory allocated	node count	ave. time per node
1	50	1000	1	0.013707	47672	100	0.000137
1	50	1000	2	0.005473	47672	100	0.000055
1	50	1000	3	0.003883	47672	100	0.000039
1	50	10000	1	0.013084	47672	100	0.000131
1	50	10000	2	0.003586	47672	100	0.000036
1	50	10000	3	0.007294	47672	100	0.000073
1	100	1000	1	0.040953	173528	198	0.000207
1	100	1000	2	0.076644	173528	198	0.000387
1	100	1000	3	0.1084	175272	200	0.000542
1	100	10000	1	0.047248	175272	200	0.000236
1	100	10000	2	0.038941	173528	198	0.000197
1	100	10000	3	0.114536	173528	198	0.000578
1	200	1000	1	0.423004	667128	398	0.001063
1	200	1000	2	2.60978	667128	398	0.006557
1	200	1000	3	3.226376	670472	400	0.008066

instance	n	coeffcients	problem #	runtime	memory allocated	node count	ave. time per node
1	200	10000	1	0.442678	670472	400	0.001107
1	200	10000	2	1.71282	667128	398	0.004304
1	200	10000	3	3.075751	770792	460	0.006686
2	50	1000	1	0.016322	47672	100	0.000163
2	50	1000	2	0.007954	47672	100	0.00008
2	50	1000	3	0.006145	47672	100	0.000061
2	50	10000	1	0.003654	48616	102	0.000036
2	50	10000	2	0.013809	85432	180	0.000077
2	50	10000	3	0.005893	47672	100	0.000059
2	100	1000	1	0.103499	199688	228	0.000454
2	100	1000	2	0.053294	173528	198	0.000269
2	100	1000	3	0.154381	196200	224	0.000689
2	100	10000	1	0.060105	182248	208	0.000289
2	100	10000	2	0.207889	175272	200	0.001039
2	100	10000	3	0.333341	175272	200	0.001667
2	200	1000	1	2.118461	1238952	740	0.002863
2	200	1000	2	3.20007	657096	392	0.008163
2	200	1000	3	13.314788	1125256	672	0.019814
2	200	10000	1	0.275241	667128	398	0.000692
2	200	10000	2	12.432164	1760616	1052	0.011818
2	200	10000	3	7.468588	670472	400	0.018671
3	50	1000	1	0.015174	85432	180	0.000084
3	50	1000	2	0.007294	47672	100	0.000073
3	50	1000	3	0.028532	212872	450	0.000063
3	50	10000	1	0.015039	47672	100	0.00015

instance	n	coeffcients	problem #	runtime	memory allocated	node count	ave. time per node
3	50	10000	2	0.012197	47672	100	0.000122
3	50	10000	3	0.016936	89208	188	0.00009
3	100	1000	1	0.458382	274680	314	0.00146
3	100	1000	2	0.618575	2564552	2940	0.00021
3	100	1000	3	0.491845	175272	200	0.002459
3	100	10000	1	0.296666	574648	658	0.000451
3	100	10000	2	1.911382	5714216	6552	0.000292
3	100	10000	3	4.581447	4278904	4906	0.000934
3	200	1000	1	27.752806	Memory limit exceeded	1	27.752806
3	200	1000	2	Timeout	4211704	1	0
3	200	1000	3	Timeout	1272664	1	0
3	200	10000	1	51.413898	Memory limit exceeded	1	51.413898
3	200	10000	2	Timeout	425480	1	0
3	200	10000	3	Timeout	213208	1	0

References

[1] Stephen A. Cook. The complexity of theorem-proving procedures. 1971.

[2] George B. Dantzig. Maximization of a linear functino of variables subject to linear inequalities: in t.c koopmans (ed.). 1947.

[3] David B. Shmoys David P. Williamson. *The Design of Approximation Algorithms*. Cambridge University Press New York, NY, USA 2011, 2011.

[4] Giacomo Zambelli Michele Conforti, Gerard Cornuejols. *Integer Programming*. Springer Publishing Company, Incorporated, 2014.

[5] Vijay Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., 2001.

[6] Laurence A. Wolsey. Heuristic analysis, linear programming and branch and bound. 1980.

[7] Laurence A. Wolsey. Mixed-integer programming. 2008.