Monash University

Final Report

Implicit enumeration using dual bounds from approximation algorithms

Nelson Frew

supervised by

Dr. Pierre Le Bodic & Arthur Mahéo

Submitted for the degree of Bachelor's of Computer Science (Honours)

November 8, 2018

I, Nelson Frew, want to know if I need this statement of originality in my paper. If so, I would like to know the conventional way to do this.

Abstract

We explore methods of using approximation algorithms in the application to the problem solving technique of implicit enumeration. Implicit enumeration refers to a divide-and-conquer problem solving strategy where one enumerates through a problem's set of solutions without finding each solution. Implicit enumeration has broad uses within many fields of Computer Science, one such example being in Mixed-Integer Programming's Branch-and-Bound, where one partially enumerates through a set of solutions by deriving subproblems and finding what are known as their dual bounds to reduce the size of the search tree. Traditionally, dual bounds are found by solving a relaxed version of the problem formulation which can be solved efficiently, however such approaches have in general no known guarantees, and may perform arbitrarily poorly on some problems. In this report, we describe our investigation into a new approach for deriving dual bounds: using approximation algorithm guarantees. We examine methods to do this, using known approximations for the 0,1 Knapsack problem as a case study, and show how we can improve our analysis to obtain higher quality bounds.

Acknowledgements Thanks everybody!

Disclaimer A component of this project was undertaken throughout a 10 week scholarship in the summer of 2017. Additionally, during the second semester of 2018 I was responsible for classes as a Teaching Associate for four groups of students.

Contents

1 Introduction

1.1 Research Context

Following the inception of Dantzig's Simplex method [13] in 1947, theoretical interest in finding optimal solutions turned to harder problems. Solving discrete optimisation problems, known as Integer Programs (IPs), became a focal point of research. Discrete optimisation gave rise to two new active research areas: exactly solving IPs, and active pursuit of polynomial time algorithms for such hard problems and their variants.

Research into optimally solving IPs led to Land and Doig's initial Branch and Bound method [30] in 1960, but had little direct applicability due to limitations in computing hardware at the time. As a response to the many discrete optimisation problems for which polynomial-time algorithms were not known, Computational Complexity Theory research began [10]. Concerns with finding polynomially bounded algorithm run times led to sacrificing optimality for efficiency, in what became heuristic and AA research through the 1970s. Approximations for the Knapsack Problem [21], Travelling Salesman Problem [7], Facility Location problem [11], and many others were devised through the decade. Until 1980, discrete problem algorithm design and AA design remained disconnected.

It wasn't until Wolsey's [47] attempt to unify the AAs of the 1970s with the Branch-and-Bound algorithm from 1960 that the fields met again. Wolsey provided a general analysis technique relating approximation worst-cases with optimal LP relaxation solutions, and devised a Branch-and-Bound procedure from these components. However, since this point, to the best of our knowledge, no further work has been done in investigating the relationship between approximations and Branch and Bound. This project investigates whether a stronger link may exist and can be leveraged to improve performance.

1.2 Background

The field of discrete optimisation [19] studies and offers solutions problems that model discrete decision making, such as scheduling, shortest path problems, or supply chain optimisation. Mixed-Integer Programming refers to a set of modelling and solving techniques for such discrete problems [48] by embedding them into what are known as Mixed-Integer Programs (MIPs) and solving them in this form. For many problems, this is the most efficient known method to find an optimal solution. The general MIP formulation, for maximisation, is as follows:

$$\begin{array}{ll} \text{maximise} & cx + hy \\ \text{subject to} & Ax + Gy \leq b \\ & x \geq 0 \text{ integral} \\ & y \geq 0 \end{array}$$

Where x and y are the sets of *decision variables* that algorithms assign values to with the purpose of maximising the *objective function* $cx + hy$. The general model we examine has objective functions that are constrained by linear inequalities, and with one or more decision variables constrained to be integer or continuous. Further, while we can also specify minimisation problems, we will consider only maximisation problems, without loss of generality, as the cases are completely analogous.

The set of decision variable values which are valid according to an optimisation problem's constraints is termed the search space. While a problem allowing continuous variables will have a continuous *feasible region* representing its search space, a problem constrained to have integer variables will have an enumerable *feasible set* of solutions. Provided with the same inequalities, because the continuous feasible region is less constrained, it will include all the values of the discrete feasible set.

Put a diagram showing continuous versus integer regions

Optimisation problems with linear constraints and continuous variables are known as Linear Programs (LPs), which are both solvable in polynomial time [3] and very efficiently in practice [13]. MIPs, on the other

hand, are characterised by having one or more variables constrained to be integer, generally resulting in an NP-Hard problem [23]. By removing the integer constraints on variables in a MIP, we obtain its associated *LP relaxation*.

Because the feasible region of continuous solutions naturally subsumes the set of feasible solutions for integers, we know that, for optimal value for the LP relaxation z_{LP} and optimal value to the MIP z:

$$z \leq z_{LP}$$

Hence z_{LP} will be referred to as an upper bound on z, or a *dual bound*. Branch-and-Bound commonly solves LP-relaxations to find dual bounds on the optimal value in order to implicitly enumerate through what are known as *primal solutions* that are feasible to the MIP. We find primal solutions by forming *subproblems*, constrained versions of the original problem, and solving the LP relaxation of these. If the solution to the LP relaxation is feasible to the original MIP, then we have found a primal solution, and so have an associated lower bound on our optimal value, which we refer to as the *primal bound*.

In order to create subproblems, we incrementally add constraints to our set of decision variables. The process of choosing a variable to constrain is known as *branching*. Branching can form at least two subproblems, which are considered children of the parent problem, where each child has a unique set of added constraints. Each of the children's LP-relaxations are then solved to obtain their own dual bounds. If we find that a given subproblem's dual bound is lower than the MIP's primal bound, then we can safely *prune* this branch as we know further evaluation and constraining of this subproblem cannot yield a better primal solution. Otherwise, if this subproblem has a dual bound above our primal bound, we continue constraining and solving subproblem relaxations until a new feasible solution is found, or a further constrained subproblem of this is pruned.

This constitutes the core of the Branch-and-Bound approach, which we formally describe below: let N_0, be the *root node* of the search tree, representing the original MIP. Then, let $ N_1, N_2, ..., N_ii, ... $ be the nodes representing every subsequent problem which may be generated from branching, which are all stored in some list structure $\mathcal{L}$. let (x^i, y^i) be x and y values associated with the optimal solution to LP relaxation LP_i for the problem at N_i, (x^*, y^*) denote an optimal solution to the MIP, $\underline{z}$ the lower bound on the optimal

value, z^* the optimal value of the MIP.

Algorithm 1: Branch-and-Bound algorithm for Mixed-Integer Programming using LP relaxations

Data: A Mixed-Integer Program.
Result: The optimal solution value, and the associated solution

1 Set $\mathcal{L} := N_0$, set $\underline{z} := -\infty$, set $(x^*, y^*) := \emptyset$;
2 **while** $\mathcal{L}$ *is not empty* **do**
3 Select a node N_i from $\mathcal{L}$, using a node selection scheme, deleting it from $\mathcal{L}$;
4 Solve LP_i to get solution value z_i and (x^i, y^i) if it exists;
5 **if** LP_i *is infeasible, i.e.* $z_i = -\infty$ **then**
6 Go to line 2;
7 **if** $z_i \leq \underline{z}$ **then**
8 Go to line 2;
9 **else if** (x^i, y^i) *is feasible to the MIP* **then**
10 Set $\underline{z} := z_i$;
11 Set $(x^*, y^*) := (x^i, y^i)$;
12 Go to step 2;
13 **else**
14 Branching: From LP_i construct 2 linear programs LP_{i1}, LP_{i2} with smaller feasible regions whose union does not contain (x^i, y^i), but contains all the solutions of LP_i with $x \in \mathbb{Z}$, by choosing a fractional variable to constrain.;
15 Add the corresponding new nodes N_{i1}, N_{i2} to $\mathcal{L}$ and go to step 2.
16 **end**
17 Set $z* := \underline{z}$;
18 Return $z*$;
19 **end**

While having an exponential worst-case time complexity, the majority of computation done in a Branch-and-Bound is in solving LP relaxations which may or may not yield a feasible solution, but always provide dual bounds. A candidate alternative, then, could be *heuristics*, which are guaranteed to yield a feasible solution in polynomial time, but these provide no way to find dual bounds. However, if a heuristic's solutions has a quality guarantee, in terms of the optimal value, it is called an *Approximation Algorithm* (AA), and we can derive duals bounds through its guarantee.

An α-approximation is an AA that guarantees that the solution value it returns will always be within a constant factor $\alpha < 1$ of the optimal solution; i.e. for an approximate solution value z_A, it must be that $\alpha z \leq z_A \leq z$. As a result, for a maximisation problem, we can analytically derive an upper bound on the optimal value from the lower bound guarantee on the optimal solution $z \leq \frac{z_A}{\alpha}$. In this way, we can use AAs in lieu of an LP relaxation to obtain valid dual bounds on our optimal solution. Because we are leveraging guarantees on the optimal value, the dual bounds we obtain are also guaranteed to be within a level of accuracy. This is a feature not found in general LP relaxations, which can provide arbitrarily poor dual bounds [8].

There are many strategies to improve a Branch-and-Bound with LPs, yet it is not necessarily possible to apply the same methods with regards to a Branch and Bound with AAs. In particular, methods of branching and node selection do not have a clear translation in this amended scheme, as the solutions provided by AAs provide different information to that of LP-relaxations. In this project, we investigate methods of effectively using dual bounds provided by AAs for the 0,1 Knapsack, as it is both a well-studied and simple to implement basis for a proof of concept. We examine the merit of this approach by looking at known approximation schemes and methods to construct high quality dual bounds from them, as well as devising branching strategies within its Branch-and-Bound.

2 Literature Review

We now give a brief literature review of the research and improvements to the Branch-and-Bound approach, as well as an overview of details and approaches with AAs that are central to the premise of this project.

2.1 Branch-and-Bound improvements and strategies

While our previous description of the Branch-and-Bound approach captured the core components, there still remain many aspects to be addressed: which variables to select for branching, alternative methods to relax a solution, and node selection schemes. These problems are in fact non-trivial and represent active research areas within Mixed-Integer Programming. In light of this, we provide an overview of known strategies and the body of research that comprises the current understanding in the field.

2.1.1 Branching Strategies

Deriving useful subproblems is central to the Branch-and-Bound approach, as it provides us with the partial enumeration required to establish high quality bounds on an optimal solution's value. When using LP relaxations, a basic strategy is to branch on the variable with the largest fractional component in the LP's solution at the current node, however [2] has shown that this can be as bad as random selection.

It is possible to make inferences based on each variable's fractional component, as shown by [43] by the proposal of "use penalties". This was used to inform the branching process where "up penalties" and "down penalties" were calculated for each variable based on their respective fractional components, after which the variable with the highest penalty in any direction was chosen. However, [34] showed with computational experiments that such approaches may have limited use, while approaches involving issuing *pseudocosts* as proposed by [5] may be more efficient in some cases. Pseudocost branching aims to choose a branching variable by creating predictions of changes in the objective function for each decision variable, from past information within the search tree. A decision variable is then chosen predicated on these predictions.

Another available strategy is *strong branching*, first introduced by [4], where again variable selection is based on a given variable's impact on the objective function [2] [1]. Specifically, the variable eliciting the most significant change in the LP-relaxation objective function for each child of a subproblem is chosen for branching.

Further work by [1] resulted in reliability branching, which combines both the concepts of strong branching and pseudocosts: for decision variables where no past information can help find a solution, we use the methods of strong branching.

A detailed survey of the literature and the current open questions in research is also provided by [35], where the reader is directed for further reading.

2.1.2 Node selection schemes

Having chosen a variable to branch on, we will have generated a number of subproblems which are represented as *nodes* at the frontier of our search tree. Because of this representation, the method used for node selection is also known as the *search strategy* for the tree.

Consider diagram here

As in our description of the Branch-and-Bound, we will refer to the set of nodes which are to be explored as $\mathcal{L}$. To continue our search, it is clear that we must choose a node to explore, however choosing a method for doing this is not necessarily straightforward. There are many methods for conducting such a tree search, the most common of which we now describe.

The *depth-first search* (DFS) strategy is a standard and well studied approach to both Branch-and-Bound and general tree traversal [17, 42]. In this strategy, one arranges the unexplored nodes in a first-in first-out (FIFO) manner, by storing $\mathcal{L}$ as a stack data structure. While this approach may seem naive, it can be modified to have very low memory costs. There are many potential drawbacks to DFS, as well as many derivations to counteract these, and the reader is directed to [35] for a more comprehensive overview of these concepts and techniques.

The *breadth-first search* (BrFS) strategy is likewise a notorious and well studied search strategy trees. The strategy employed by BrFS is often considered to be opposite to DFS, as it uses a last-in first-out (LIFO) search, often by storing $\mathcal{L}$ in a queue data structure. Despite being a well known strategy, BrFS is not often used within Branch and Bound schemes.

The final main category of search strategy we will consider is known as the Best-First Search (BFS). As its name indicates, instead of having the order which nodes are created govern the order which they are chosen, we define a way to appraise the "quality" of nodes to inform our choice. One can conduct such a search by storing $\mathcal{L}$ as a priority queue based on the "quality" of a node. The definition of "quality" is clearly important: within a Branch-and-Bound a basic but useful indicator could be any information about a node's dual bound. BFS is known to be highly effective; [14] showed that BFS could provably explore the lowest number of subproblems in certain families of scenarios, although it is also relevant to note that [41] demonstrated potential flaws in the scheme.

2.1.3 Bounding optimal solutions

Using LP relaxations to find dual bounds is a standard method used within Branch-and-Bound schemes. Solving LPs has been efficiently possible since the introduction of Dantzig's Simplex method [13]. Despite its efficiency, LP relaxations can provide arbtirarily poor bounds in this context: for Graph Colouring, [8] presented a formulation where the linear relaxation is significantly weak. However, this is only one of many known ways to obtain valid dual bounds within Mixed Integer Programming: other examples include Semidefinite Programming relaxations (see [32, 44]), and Langrangian relaxations [16].

2.1.4 Other extensions, improvements, and related techniques

State-of-the-art MIP solvers often combine Branch-and-Bound strategies with various extensions to improve performance. One common extension to the method includes using cutting planes [18] to produce the Branch-and-Cut algorithm. The cutting plane approach, introduced by [12], and extended to general Integer Programs (IPs) in [18], involves solving the LP relaxation of a program and formulating a constraint, called a cutting plane, that separates this LP solution from the rest of the search space. A Branch-and-Cut approach [37, 38], then, is a Branch-and-Bound with the ability to choose to add a cutting plane instead of branching on variable. For a more detailed treatment on this topic, the reader is referred to [9]. Other key methods for extending the Branch-and-Bound involve presolving techniques [33] primal heuristics [6].

Another related concept is the idea of warm starting a solution in linear programming linked to integer programming by [39]. In short, warm starting is a method to exploit information gained about the problem from previous computations in order to inform future ones. One method for this is to use previous computation to find a starting primal bound in a Branch-and-Bound.

2.2 Approximation algorithms

2.2.1 General approximations

2.2.2 Approximating the 0,1 Knapsack Problem

The case study chosen for experimentation within this project is the 0,1 Knapsack Problem (KP). Known to be NP-Complete [27], the KP has been studied extensively both in solving optimally and approximately (see,

for example [28, 20, 22, 29, 15, 36]), and as such, it serves well as a basis for a proof of concept. We describe the formulation for KP as follows: given n items with associated weights w_i and values v_i, and Knapsack capacity W, our program is

$$\begin{aligned} \text{maximise} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq W, \text{ and } x_i \in \{0,1\} \end{aligned}$$

It is well known that the KP lends itself well to a Dynamic Programming (DP) approach, although there are many ways to conduct this [26]. One of the most common examples, performing a DP by values as presented by [45], can find a solution in $O(n^2P)$ time, where P is the value of the most profitable item, i.e. $P = \max_i v$. Let $A(i, p)$ be the minimal weight of the solution to the KP with only the first i items available, where the total value is exactly p. If no such set exists, that is, the first i items cannot yield a value of p and so cannot have an associated minimal weight, then $A(i, p) = \infty$.

The DP recurrence can then be defined as follows:

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

The optimal solution value can then be found by finding the value $\max\{p \mid A(n,p) \leq W\}$.

Being NP-Hard, the KP does not admit a polynomial time algorithm; this DP algorithm runs in fact in *pseudopolynomial* time.

Definition 1. *An algorithm runs in *pseudopolynomial time* if it runs in time bounded by a polynomial of the input size, when the numerical component of the input is encoded in unary.*

The numeric component of our time complexity is P, which in this case is the source of the pseudopolynomiality. To obtain an approximation, we will show that one can scale this value P until it is bounded by a polynomial in n while maintaining a quality guarantee. A *polynomial Time Approximation Scheme* (PTAS) has this characteristic.

Definition 2 (PTAS)**.** *A Polynomial-Time Approximation Scheme is an algorithm guaranteed to give a solution value within a factor of* $(1-\epsilon)$ *(for maximisation problems) of the optimal value, and have a running time bounded by a polynomial in the input size, for fixed values of* ϵ*.*

A classic example of a PTAS was presented by Sahni [40], which has a time complexity of $O(n^{1/\epsilon})$.

While the run time of a PTAS is indeed bounded by a polynomial in n, the complexity can be exponential in terms of $\frac{1}{\epsilon}$, so long as $\frac{1}{\epsilon}$ is fixed. If we restrict the definition of a PTAS further, then, so that the running time is bounded by a polynomial in both n and ϵ, we obtain a *Fully Polynomial Time Approximation Scheme* (FPTAS). This means we can choose an ϵ that gives us a high quality solution while still being polynomially bounded.

Definition 3 (FPTAS)**.** *A Fully Polynomial Time Approximation Scheme is an algorithm guaranteed to give a solution value within a factor of* $(1-\epsilon)$ *(for maximisation problems) of the optimal value, and to have a running time bounded by a polynomial in the input size and* $\frac{1}{\epsilon}$*.*

It is common for FPTAS's for the KP to achieve polynomially-bounded run time by reducing the input problem's complexity, and using a DP algorithm to solve this simplified problem. As such, the construction of an FPTAS often involves two distinct steps: a preprocessing step to form the simpler problem, and a solving step to find an approximate value for the original problem. We consider the examples from the literature which have led to what are now textbook examples of FPTAS's, followed by a description of the state-of-the-art method.

2.2.3 Ibarra and Kim's FPTAS (1975)

What is now a standard method of constructing an FPTAS for the KP was first described by Ibarra & Kim [21]. While there are several components at play in their proposed algorithm, the core of what creates the approximation involves scaling the values of all items down to by a factor dependant on ϵ. The distillation of their algorithm, as described by [45], is presented below:

Algorithm 2: FPTAS for Knapsack

Data: A KP instance, and error parameter ϵ.
Result: The weight and value pair of the optimal solution
1 Given ϵ, let $K = \frac{\epsilon P}{n}$;
2 For each object i, define adjusted values $v' = \lfloor \frac{v_i}{K} \rfloor$;
3 Run a DP with these adjusted item values;
4 Return the approximated solution set, S', provided by the DP;

By using the DP recurrence (1) described in the previous section, we can obtain an FPTAS for the Knapsack. We now present the proof for this as described by [45].

Lemma 1. *For the item set returned by the approximation, S', its associated approximated value $z_{S'}$, and value of the optimal solution set z_O, the following inequality holds:<!–[TODO] Pierre: Should we just keep z* AND z' S' is a bit confusing –>*

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

Proof. Let $z_{S'}$ be the value of the 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, the FPTAS's solution set S', and z'_O be the optimal solution set with similarly adjusted value for all items.

Because of the flooring operation truncating off at most K, we know that the difference between z_O and $K \cdot z'_O$ would be at most nK. Furthermore, since $z'_{S'}$ is the optimal value for the adjusted profits, z'_O cannot exceed it.

So,

<!–[TODO][CLARIFY] Pierre: All these divets under each inequality–>

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

Since $K = \frac{\epsilon P}{n}$, and $P \leq z_O$:

$$z_O - nK = z_O - \epsilon P \geq z_O - \epsilon z_O = (1 - \epsilon) \cdot z_O$$

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

□

We show that this must be an FPTAS as it was described by [45]:

Theorem 1. *Algorithm 2 is an FPTAS for KP*

Proof. By Lemma 1, we know that the solution found will be within a factor $(1 - \epsilon)$ of the optimal value z_O. We know that the running time of the original DP algorithm was $O(n^2 P)$, so with our scaled profits our run time reduces to

$$O\left(n^2 \cdot \left\lfloor \frac{P}{K} \right\rfloor\right) = O\left(n^2 \cdot \left\lfloor \frac{n}{\epsilon} \right\rfloor\right)$$

which is bounded by a polynomial in n and $\frac{1}{\epsilon}$. □

Because we reduced P down to be bounded by n, our time complexity now reduces to be $\mathrm{O}(n^3/\epsilon)$ rather than $O(n^2P)$. This constitutes the core of what is one of the textbook approaches to constructing an FPTAS for KP. Because our focus will reside on the guarantees provided by this preprocessing step, we omit further refinements to this algorithm which accompanies the original description given by [21].

2.2.4 Lawler (1978)

The second standard procedure for obtaining an FPTAS for the KP was originally presented by [31]; we present it as described by [46]. Building on the contributions of [21], this approach describes an alternative FPTAS which still uses the preprocessing shown in Algorithm 2. Specifically, we construct this algorithm by changing the DP approach.

For an integer $j \leq n$, let A_j be an array which contains representations of solutions to KP for the first j items. A solution is represented by a tuple $(v, w) \in A_j$ if there is a set of the first j items with value v and weight $w \leq W$. Instead of explicitly storing all partial solutions, we resolve to only store partial solutions which are not *dominated* by other partial solutions. We say a solution (v, w) dominates solution (v', w') if $v \geq v'$ and $w \leq w'$.

The DP algorithm, as presented by [46], then follows:

Algorithm 3: Alternative dynamic programming for Knapsack

Data: A KP problem instance
Result: The value of the optimal solution

```
for j ← 2 to n do
    for each (v, w) ∈ A_j do
        if w + w_j ≤ W then
            Add (v + v_j, w + w_j) to A_j
        end
    end
    Remove dominated pairs from A_j
end
return max_{(v,w)∈A_j} v
```

It can be shown that this algorithm correctly finds the optimal value to the KP, and the reader is directed to either [31] or [46] for the proof of this result. By using this DP algorithm in Algorithm 2, we obtain another valid FPTAS for KP, which has a running time again of $O(n^3/\epsilon)$.

The original version of this algorithm, presented in [31], contains further subtle refinements to this algorithm by improving details of the FPTAS provided by [21].

2.2.5 Kellerer and Pferschy (2004)

The final case of an FPTAS for KP which we will review was presented by Kellerer and Pferschy in 1999 in [24] and improved upon in a follow up paper in 2004 in [25]. At the time of writing, the FPTAS presented in the latter paper is the state of the art. The FPTAS presented operates on a DP scheme designed to exploit characteristics rendered by the proprocessing step. The authors introduce an auxiliary problem, which they then devise an efficient algorithm for, before using this problem as a generalised formulation of the KP.

The preprocessing step proposed involves making a distinction between items with "large" and "small" profits. All items with large profits are then organised into equal length intervals of value, each of which containing a set of subintervals whose lengths increase with the values defining them; i.e. a higher profit subinterval will have a larger range. Then, only the first m smallest weight items of a given subinterval are kept, according to the given subinterval and the error parameter ϵ. These remaining items then have all their profits reduced to the lower bound of their respective subinterval, completing the preprocessing step.

In order to complete this procedure, we compute a lower bound $\underline{z}$ on the optimal value such that

$$\underline{z} \leq z \leq 2\underline{z}$$

and modify our ϵ value as follows:

$$\hat{\epsilon} := \frac{1}{\lceil \frac{2}{\epsilon} \rceil}$$

to make both $\frac{1}{\epsilon}$ and $\frac{1}{\epsilon^2}$ integer values, a fact exploited in the following DP scheme. The optimal solution value of this adjusted set of large profit items L is at worst $(1-\epsilon)$ of the optimal value for the unadjusted profit set $\mathcal{L}$:

Theorem 2. $z_L \geq (1-\epsilon) z_{\mathcal{L}}$

Proof. (also make sure that I have introduced everything to make sure this proof succinctly makes sense). □

[TODO] PROVE ME

Having established this, the next object of interest is the proposal DP scheme that will exploit this scaled and reduced item set. Central to achieving fast run time is providing a solution to what is named the *Vector Merging* (VM) problem: for vectors $A = (A_1, ..., A_n)$, and $B = (B_0, ..., B_{n-1})$, we want to compute a vector C defined by:

$$C_k := \min \left\{ A_l + \sum_{j=0}^{k-l} B_j \;\middle|\; l = 1, \ldots, k \right\}, k = 1, \ldots, n$$

While this problem is trivially solvable in $O(n^2)$ time, we can solve this problem fully in $O(n\log n)$ time. The basis for this algorithm comes from a series of observations, which we now describe.

The value for each C_k is determined by the value l which minimises the sum of A_l and its associated B entries. We say that C_k *originates* from l, and describe this with the "origin" vector entry $origin[k] = l$. It will likely be the case that *origin* will have a large number of consecutive values l that are identical; therefore we can store just the indices where a value of origin starts and ends a consecutive streak. We can then conduct a binary search in the range specified by these start and end points to find the point where a given l no longer minimises a given C_k. We then repeat this process for all k for all remaining entries of C, producing n iterations, each with a binary search, resulting in a time complexity of $O(n\log n)$.

To relate this to the KP, we first define a DP approach. The DP scheme that we employ maintains an array $y_j[q]$, which contains the minimum weight solution for the first j items with profit q. Because we can evaluate various profit values q of $y_j[q]$ in any order, and because we preprocessed our profits such that each profit in subinterval t has profit p_t, we simultaneously solve the entries with the same residual value r from the division $\frac{q}{p_t}$, for each $r = 0, \ldots, p_t - 1$. We can then solve $y_j[r + p_t]$, for every multiple of p_t up to q simultaneously.

In addition, we can also consider all m items which have this same value p_t simultaneously, denoted as $j+1, j+2, \ldots, j+m$, allowing us to use the recurrence:

$$y_{j+1}[2 \cdot p_t] := \min\{y_{j-1}[2 \cdot p_t], y_{j-1}[p_t] + w_1^t, y_{j-1}[0] + w_1^t + w_2^t\}$$

Which has the exact same structure as our problem VM, where we let A take the values of the profits, B take the values of the weights, and C be the DP array. The authors have shown that this is indeed a valid FPTAS which runs in $O(n\log(1/\epsilon) + 1/\epsilon^3\log^2(1/\epsilon))$ time. Other details, such as deriving the optimal item set as well as the optimal value are also provided, which we omit here for brevity. As well as the original paper, the authors also provide an overview of the algorithm in detail in chapter 6 of [28], where the reader is also referred for further information.

2.3 Approximations, Linear Programming and Branch-and-Bound

All that now remains to review is the existing literature which addresses AAs in an implicit enumeration setting. As far as we are aware, the only focused treatment of this was presented by Wolsey [47].

Acknowledging that LPs are tightly knitted to the state of the art in Branch and Bound, Wolsey demonstrated the use of a framework which allows one to emulate characteristics of linear duality, the basis for dual bounds with LPs, with AAs. In this way, the first and only provided method for finding dual bounds with AAs was presented.

For an approximate solution value z_A, an LP relaxation solution value z_{LP}, Wolsey showed we can construct inequalities of the form:

$$z_A \leq r \cdot z_{LP} + s$$

for $r \geq 1$ and constant s. Having provided general way to find the worst-case result for z_A in terms of z_{LP} he demonstrated that we can construct a Branch-and-Bound where the value of z_A would monotonically increase as enumeration progressed, inevitably converging to optimality.

Wolsey then showed that we can extend this analysis to formulations on the Bin Packing Problem, Longest Undirected Hamiltonian Tours, Minimum Length Eulerian Tours and the Chinese Postman Problem. This is the only implicit enumeration scheme leveraging AA guarantees that we are aware of.

3 Design, Analyses, and Implementation

Given the examples from the literature to Branch-and-Bound and AAs, we will now briefly review the research questions motivating our work.

TODO FORMAT THESE NICER

Can AAs be used within Branch-and-Bound to efficiently solve combinatorial problems to optimality?

The key word in this question being "efficiently." We know that it is possible to achieve implicit enumeration already, however we do not know the extent to which this may be a competitive option. In the interest of this goal, a relevant subquestion arises:

How can we improve solution quality analyses for given AAs to find high-quality dual bounds?

Indeed, while we do have a viable dual bound for any α-approximation, it is nevertheless in our interest to find improvements to this. Such a question relates primarily to the analysis of the guarantees for a given existing AA, and so another question arises:

How can we devise approximation schemes which are tailored towards the computation of high-quality dual bounds?

By designing a preprocessing procedure with dual bounds in mind, we may be able to then analytically derive stronger dual bounds from these.

We will now show how we can derive dual bounds from the analysis of the guarantees which define a given FPTAS, and present a branching strategy in light of this. Then, we will improve the bounds obtained with further analytical insights, and then how we can preprocess an instance in the interest of finding dual bounds.

3.1 Analytic derivation of dual bounds *a priori*

To form a basis to our discussion, we reintroduce the bounds obtained by the FPTAS originally presented by Ibarra and Kim in [21]. In particular, recall that we found

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

By basic algebraic manipulation, we can then obtain the following bounds for the optimal value *a priori*, i.e. before seeing the solution using information based on the worst-case analysis:

$$z_O \in [z_{S'}, z_{S'} + nK]$$

which provides us with a method to bound our subproblems, leaving the issue of branching to be addressed. In devising a branching strategy, we must consider what variables we can constrain which will help us minimise our dual bounds. Given that our upper bound on z_O is determined by the scaled values of the approximation, we can see that the value truncated by the preprocessing step presented in section 2.2.1 determines the quality of our dual bound.

Insert visualisation of it

We can attempt to minimise our dual bounds, then, by attempting to maximise the difference between $z_{S'}$ and $K \cdot z'_{S'}$, which we can do by first branching on variables which truncate the most value in the scaling step. For the remainder of this report, this strategy will be termed *Truncation branching*.

This provides us with the following first attempt at an amendeded Branch and Bound algorithm:

Algorithm 4: Branch-and-Bound with Approximation Algorithms

Data: A MIP instance, and an AA for the given problem.
Result: The weight and value pair of the optimal solution

```
1  Let z̄ := approximate solution;
2  while L is not empty do
3  |  Choose a node N_i from L, and delete it from L;
4  |  Run the AA, deriving upper bound (UB) and lower bound (LB) in the process;
5  |  if UB ≤ z̄ then
6  |  |  Go to line 2;
7  |  else
8  |  |  if LB> z̄ then
9  |  |  |  z̄ := LB;
10 |  end
11 |  else if for N_i's parent's UB, UB_p, UB > UB_p then
12 |  |  Set UB := UB_p
13 |  else
14 |  |  Choose a variable to branch on, then generate and enqueue new child nodes;
15 |  end
16 |  Return z;
17 end
```

Implementing this approach, the average relative decreases for node counts and dual bound values are shown in table 1.

	$n = 50$			$n = 100$			$n = 200$		
	DP	Nodes	DB	DP	Nodes	DB	DP	Nodes	DB
Random branching									
Inst #1 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #2 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #3 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #1 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #2 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #3 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Truncation branching									
Inst #1 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #2 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #3 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #1 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #2 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #3 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666

Table 1: Comparison between the a priori bound and use of standard LP bounding. Values are relative decrease (percentage)

3.2 Derivation of dual bounds *a posteriori*

We can obtain improvements to our dual bounds by observing information obtained following completion of the FPTAS, i.e. *a posteriori*. To do this, we first observe that the approximate solution set S' will have a profit of $z_{S'}$. This value will be the value of the adjusted profits $z'_{S'}$ scaled up by K, plus any profit ω lost from the flooring operation, i.e.

$$\begin{aligned} p(S') &= Kz'_{S'} + \omega \\ &\geq (1-\epsilon) \cdot z + \omega \end{aligned}$$

Note that in our *a priori* bound, ω was unknown and the analysis treated the worst case where $\omega = 0$. However, after completing our FPTAS, ω becomes known and so we obtain the stronger result

$$\begin{aligned} (1-\epsilon) \cdot z + \omega &= z - \epsilon z + \omega \\ &\leq z - \epsilon P + \omega \\ &= z - Kn + \omega \end{aligned}$$

Further, having run the FPTAS, we know the value of $z'_{S'}$. Since $Kz'_{S'}$ could only have lost at most nK of precision,

$$\begin{aligned} z_{S'} \geq z_O - Kn + \omega \implies z_O \leq z_{S'} + Kn - \omega \\ z_O \in [z_{S'}, K \cdot z'_{S'} + Kn - \omega] \end{aligned}$$

Implementing this approach, we can see how this performs on given instances of KP in table 2.

	$n = 50$			$n = 100$			$n = 200$		
	DP	Nodes	DB	DP	Nodes	DB	DP	Nodes	DB
Truncation branching									
Inst #1 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #2 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #3 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #1 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #2 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #3 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666

Table 2: Comparison between a posteriori bound and use of standard LP bounding. Values are relative decrease (percentage)

	$n = 50$			$n = 100$			$n = 200$		
	DP	Nodes	DB	DP	Nodes	DB	DP	Nodes	DB
Truncation branching									
Inst #1 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #2 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #3 (WS)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #1 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #2 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666
Inst #3 (VV)	0.123	0.456	0.789	0.111	0.222	0.333	0.444	0.555	0.666

Table 3: Comparison between a posteriori bound and use of standard LP bounding. Values are relative decrease (percentage)

3.3 Improved *a posteriori* dual bounds

To improve our approach further, we can specifically design a preprocessing scheme intended for finding dual bounds. An approach to do this is by modifying our value-scaling procedure to round all the profits up, instead of rounding down. That is for every value v_i, we define its adjusted value as

$$v_i'' = \left\lceil \frac{v_i}{K} \right\rceil$$

This results in some amount of added values when we scale back up by K, which we use ω to denote. We can show that this still adheres to our required $(1 - \epsilon) \cdot z_O$ lower bound: let S'' be the approximate set where profits are rounded up, and $z''_{S''}$ be the value of S'' with scaled down, rounded up profits.

$$\begin{aligned} z_{S''} &= K z''_{S''} - \omega \\ &\geq z_O - \omega \\ &\geq z_O - K|S''| \\ &\geq z_O - nK \\ &= z_O - \epsilon P \\ &\geq (1 - \epsilon) z_O \end{aligned}$$

Implementing this approach, we can see how this performs on given instances of KP in table 3.

4 Future work

While we have given a preliminary treatment of the use of using a posteriori knowledge to find dual bounds with AAs, much remains to be done. Further work investigating the integration of AAs into a Branch-and-Bound, such as the potential of warm-starting DP solutions, as is done with LP, may be worthwhile.

In addition, the algorithms we conducted tests on in this report did not include the current state of the art algorithm, and as such had a limited capacity to handle instances with large amounts of items. Specifically, our FPTAS ran in $O(n^3/\epsilon)$ which limited our ability to handle instances with high amounts of items. The state of the art algorithm presented in the literature review does not have this growth in n, and so may be better equipped to handle larger instances quickly.

It will, of course, be worth investing time in investigating the viability of our proposed method with known approximations to more complex problems, such as Christofides' [7] approximation for the Metric Travelling Salesman problem.

References

[1] Tobias Achterberg. *Constraint integer programming*. PhD thesis, 2007.

[2] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[3] Farid Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM journal on Optimization*, 5(1):13–51, 1995.

[4] Bixby Applegate and R Bixby. Chvátal & cook . finding cuts in the tsp. Technical report, Technical report, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS). DIMACS Technical Report 95-05, 1995.

[5] Michel Bénichou, Jean-Michel Gauthier, Paul Girodet, Gerard Hentges, Gerard Ribière, and O Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971.

[6] Timo Berthold. Primal heuristics for mixed integer programs. 2006.

[7] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[8] Pablo Coll, Javier Marenco, Isabel Méndez Díaz, and Paula Zabala. Facets of the graph coloring polytope. *Annals of Operations Research*, 116(1-4):79–90, 2002.

[9] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. Integer programming, volume 271 of graduate texts in mathematics, 2014.

[10] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[11] Gerard Cornuejols, Marshall L Fisher, and George L Nemhauser. Exceptional paper—location of bank accounts to optimize float: An analytic study of exact and approximate algorithms. *Management science*, 23(8):789–810, 1977.

[12] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[13] George B Dantzig. Maximization of a linear function of variables subject to linear inequalities. *New York*, 1951.

[14] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a. *Journal of the ACM (JACM)*, 32(3):505–536, 1985.

[15] RS Garfinkel and GL Nemhauser. Integer programming, 1972.

[16] Arthur M Geoffrion. Lagrangean relaxation for integer programming. In *Approaches to integer programming*, pages 82–114. Springer, 1974.

[17] Solomon W Golomb and Leonard D Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, 1965.

[18] Ralph E Gomory et al. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical society*, 64(5):275–278, 1958.

[19] PL Hammer, EL Johnson, and BH Korte. Discrete optimization, vol i and ii, 1979.

[20] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21(2):277–292, 1974.

[21] Oscar H Ibarra and Chul E Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)*, 22(4):463–468, 1975.

[22] Giorgio P Ingargiola and James F Korsh. Reduction algorithm for zero-one single knapsack problems. *Management science*, 20(4-part-i):460–463, 1973.

[23] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[24] Hans Kellerer and Ulrich Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3(1):59–71, 1999.

[25] Hans Kellerer and Ulrich Pferschy. Improved dynamic programming in connection with an fptas for the knapsack problem. *Journal of Combinatorial Optimization*, 8(1):5–11, 2004.

[26] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Basic algorithmic concepts: Dynamic programming. In *Knapsack problems*, pages 20–27. Springer, 2004.

[27] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Introduction to np-completeness of knapsack problems. In *Knapsack problems*, pages 483–493. Springer, 2004.

[28] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Knapsack problems. 2005.

[29] Peter J Kolesar. A branch and bound algorithm for the knapsack problem. *Management science*, 13(9):723–735, 1967.

[30] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[31] Eugene L Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4(4):339–356, 1979.

[32] László Lovász and Alexander Schrijver. Cones of matrices and set-functions and 0–1 optimization. *SIAM journal on optimization*, 1(2):166–190, 1991.

[33] Ashutosh Mahajan. Presolving mixed-integer linear programs. *Preprint ANL/MCS-P1752-0510, Mathematics and Computer Science Division*, 2010.

[34] Gautam Mitra. Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming*, 4(1):155–170, 1973.

[35] David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016.

[36] George L Nemhauser and Zev Ullmann. Discrete dynamic programming and capital allocation. *Management Science*, 15(9):494–505, 1969.

[37] Manfred Padberg and Giovanni Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1–7, 1987.

[38] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[39] TK Ralphs and M Güzelsoy. Duality and warm starting in integer programming. In *The Proceedings of the 2006 NSF Design, Service, and Manufacturing Grantees and Research Conference*. Citeseer, 2006.

[40] Sartaj Sahni. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)*, 22(1):115–124, 1975.

[41] Edward C Sewell and Sheldon H Jacobson. A branch, bound, and remember algorithm for the simple assembly line balancing problem. *INFORMS Journal on Computing*, 24(3):433–442, 2012.

[42] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[43] JOHN A Tomlin. Branch and bound methods for integer and non-convex programming. *Integer and nonlinear programming*, 1:437–450, 1970.

[44] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM review*, 38(1):49–95, 1996.

[45] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[46] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[47] Laurence A Wolsey. Heuristic analysis, linear programming and branch and bound. In *Combinatorial Optimization II*, pages 121–134. Springer, 1980.

[48] Laurence A Wolsey. Mixed integer programming. *Wiley Encyclopedia of Computer Science and Engineering*, pages 1–10, 2007.