Modern Algorithms for Integer Linear Programming

Trevor Henderson, Roger Jin, Lucia Li

November 21, 2018

1 Introduction

This paper surveys new results on the complexity of Integer Linear Programming (ILPs) from Eiesenbrand et al. [3] and Jansen et al. [5]. Their work is the first to improve on a classic result from Papadimitriou [6] that is now almost thirty years old.

In section 2, we describe Papadimitriou's algorithm, which introduces a relation between ILPs and the longest path problem. Using this intuition, Papadimitrious develops an algorithm that runs in time $O\left(\Delta^m \cdot n^{2m+2} \cdot (m\Delta + m\|b\|_\infty)^{(m+1)(2m+1)}\right)$. In section 3, we show how Eisenbrand et al. use the Steinitz lemma from linear algebra to shrink the search space for the longest path, significantly reducing the complexity to $n \cdot O(m\Delta)^{2m} \cdot \|b\|_1^2$. In section 4, we illustrate Jansen et al.'s algorithm which extends Eisenbrand et al's results using dynamic programming as well as other techniques to achieve a runtime of $O(m\Delta)^{2m} + \text{LP}$, where LP is the time needed to compute a linear program, which is suspected to be almost optimal. In section 5, we discuss how these algorithms can acheive state of the art results on the unbounded knapsack problem.

1.1 Integer Linear Programming

Linear programming is a beautiful, general framework that encapsulates many fundamental problems of computer science; shortest path, max flow, and zero-sum games among others. Linear programs can be solved in polynomial time and efficiently in practice. Formally, a linear program is defined by its constraints, $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$, as well as its objective $c \in \mathbb{R}^n$:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

Integer linear programming differs by the added constraints that $A \in \mathbb{Z}^{m\times n}, b \in \mathbb{Z}^m, c \in \mathbb{Z}^n$, and $x \in \mathbb{Z}^n_{\geq 0}$. This subtle difference makes ILPs NP-Complete.

However, if we consider the number of constraints to be fixed, then ILPs can be solved in polynomial time. This makes the following algorithms *pseudopolynomial*.

Additionally, the algorithms will depend on the magnitude of the constraints. We will define Δ to be a bound on the absolute value of the entries of A:

$$\Delta = \max_{i,j} |A_{ij}|$$

Like linear programs, ILPs generalize a number of important problems like planning and scheduling. Even though the algorithms we describe are pseudopolynomial, they are useful for certain problems with a small, fixed number of constraints.

For example, recall the unbounded knapsack problem. Given a set of n types of items, i, with values $c_i \in \mathbb{Z}$ and a weights $A_i \in \mathbb{Z}$, determine the quantity of each item $x_i \in \mathbb{Z}_{\geq 0}$ that we should put into our backpack to maximize the total value without exceeding the weight limit $b \in \mathbb{Z}$.

This corresponds to the ILP:

$$\max \quad \sum_{i=1}^{n} c_i x_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} A_i x_i = b \tag{2}$$

$$x \in \mathbb{Z}^n_{\geq 0} \tag{3}$$

Notice that in the knapsack problem, the matrix A is actually of shape $1 \times n$. Since the number of rows, m, is fixed, the knapsack problem can be solved efficiently using the following algorithms as described in section 5.

2 Papadimitriou's Algorithm

In 1981, Papadimitriou introduced an analogy between ILPs and the longest path problem which serves as the inpiration for his algorithm [6]. To illustrate this, realize that the constraint $Ax = b$ is equivalent to saying that b is a weighted sum of the column vectors of A:

$$b = x_1 A_1 + \ldots + x_n A_n \tag{4}$$

With this expansion, we can begin to think about a feasible solution $Ax = b$ as a path. This path begins at the origin and makes n steps arriving at b as shown in figure 1. Each step of the path is a column A_i scaled by a positive integer, x_i. To add the objective, $\max c^T x$, we simply need to weight each edge corresponding to the ith step with cost $c_i x_i$. The longest path (according to edge weight, not physical distance) from the origin to b is the optimum of the ILP.

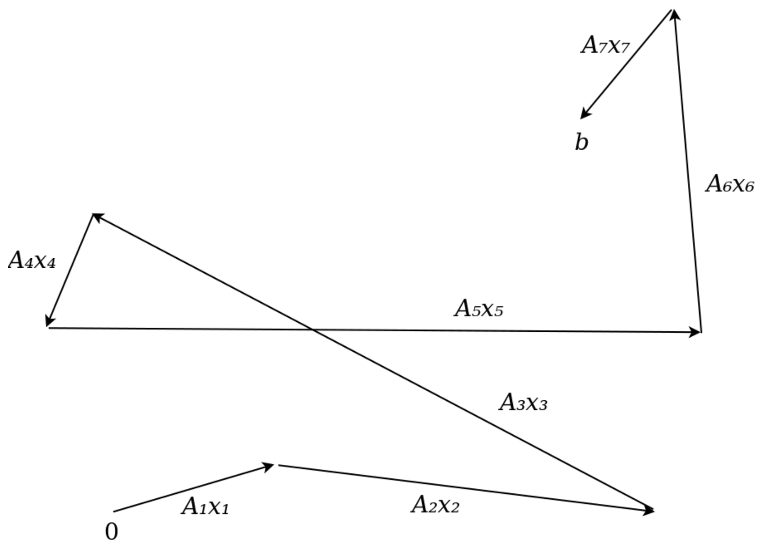

Figure 1: The path in Papadimitriou's algorithm from the origin to b, composed of steps $A_i x_i$.

Now, it remains to actually construct the graph that we are going to search over. Clearly the search space is enormous and it is this issue that Eisenbrand et al. realized they could improve in their following paper. Nevertheless, Papadrimitriou's "brute force" construction still achieves pseudopolynomial time.

The graph is going to be composed of $n + 1$ layers, the result of n steps. The first layer will contain the origin, and each subsequent layer is formed by adding all possible values of $x_i A_i$ to the layer that precedes it. Specifically, for every node, $(i - 1, v)$, representing a node in layer $i - 1$ corresponding to point v, we construct nodes $(i, v + x_i A_i)$ for any feasible value of x_i and connect the nodes with a directed edge that has weight $c_i x_i$.

To define how immense this graph is, we will introduce the value B, which will bound the maximum value of any entry of the optimal solution, $\|x^*\|_\infty \leq B$. For any particular point v, if v was constructed by adding the largest possible values of $A_i x_i$ for all n steps, that node could be at most $\|v\|_\infty \leq n\Delta B$. Therefore we can bound the number of nodes in each layer by $(n\Delta B)^m$.

Since the graph is layered we can solve the longest path problem using dynamic programming. Each node has at most B children, so computing longest paths in the next layer takes $O(B(n\Delta B)^m)$ time. Since this update process must be done for each of the n layers, the total running time is $O(\Delta^m (nB)^{m+1})$. Papadimitriou derives that $B = O\left(n(m\Delta + m\|b\|_\infty)^{2m+1}\right)$, which gives us the claimed running time:

$$O\left(\Delta^m n^{2m+2}(m\Delta + m\|b\|_\infty)^{(m+1)(2m+1)}\right)$$

3 Eisenbrand and Weismantel's Algorithm

Eisenbrand and Weismantel's algorithm [3] improves upon Papadimitriou's algorithm [6] after a gap of almost three decades. Their algorithm also views ILPs as a longest path problem, but they construct the graph in a different way to make it much smaller. This reduction relies on Steinit'z lemma from linear algebra to bound the space of feasible solutions. Once the graph has been constructed, the solution is found with common search algorithms.

First we will describe Steinitz Lemma in section 3.1. Then, we will discuss how we can use this to obtain a feasible solution in section 3.2. With little additional work, we can extend this to an optimal solution as described in section 3.3.

3.1 Steinitz Lemma

Eisenbrand et al.'s work as well as Jansen et al's work that follows [5] rely on the Steiniz lemma from linear algebra. The lemma says that there exists a way to rearrange a set of sufficiently small vectors whose sum is zero, so that each partial sum of vectors is bounded.

We can think of this sum as a path from the origin to itself where each vector represents a step. The lemma says that the steps of this path can be reordered so that they are always relatively close to the origin as shown in figure 2.

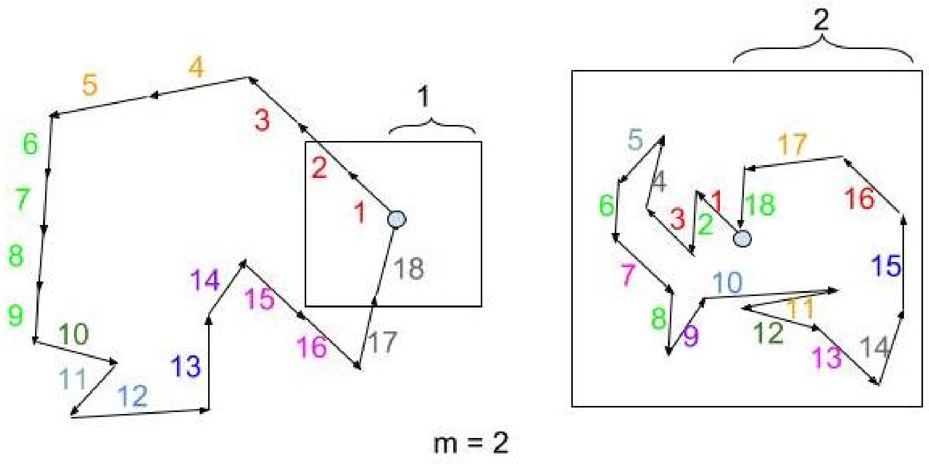

Figure 2: A visualization of the Steinitz lemma for in 2 dimensions. The large path on the left is reordered to form more compressed path on the right with an ℓ_∞ norm less than $m = 2$. Steps of the same size are the same color on both sides.

Formally, the Stieiniz lemma is given as follows:

Theorem 1 (Steinitz)**.** *Let $w_1, \dots w_n \in \mathbb{R}^m$ such that:*

$$\sum_{i=1}^{n} w_i = 0 \quad \textit{and} \quad \|w_i\| \leq 1 \textit{ for all } i.$$

There exists a permutation $\pi \in S_n$ such that all partial sums satisfy

$$\left\| \sum_{j=1}^{k} w_{\pi(j)} \right\| \leq m \textit{ for all } k = 1, \dots, n \tag{5}$$

Note that if $\|w_i\|$ is bounded by some other constant, $\|w_i\| \leq \delta$, then through scaling the partial sums are bounded by δm.

3.2 Feasibility

In Papadimitriou's algorithm we formulated ILP as a path planning problem by first realizing that the constraint $Ax = b$ is equivalent to saying that b is a weighted sum of the column vectors of A as in equation (4). Because each element, x_i, is an integer we can continue this expansion and just consider b to be the sum of some number of each column of A:

$$b = \underbrace{A_1 + \ldots + A_1}_{x_1 \text{ times}} + \underbrace{A_2 + \ldots + A_2}_{x_2 \text{ times}} + \ldots + \underbrace{A_n + \ldots + A_n}_{x_n \text{ times}} \tag{6}$$

Expanded like this, we can think about any feasible solution $Ax = b$ as another path; we begin at the origin and make steps equal to column vectors of A until we reach b. This contrasts with Papadimitiou, whose steps are column vectors of A *times integers* x_i. While Eisenbrand et al.'s path is going to have many more steps, they will not be divided into layers like Papadrimitriou's. This gives us the freedom to do some reorganization using Steiniz's lemma to shrink the search space.

In particular, our search must extend to any vector that is on the path from the origin to b. These vectors are partial sums of the column vectors in equation (6). The Steinitz lemma lets us bound these partial sums. Note that equation (6) already holds two important properties needed to apply the Steinitz lemma:

1. The steps in the path can be made in *any order*. Despite being written in column order there is no reason for this, its just addition.

2. Each step is *small*. Recall that we are assuming that the absolute value of each element of A is bounded by Δ.

The one missing piece is that our path starts from the origin and ends up at b, where as Steiniz's lemma holds for paths that start at the origin and end up back at the origin. We can fix this through some rearrangement. Before doing that, let's simplify the notation. Let t be the number of steps in the path and let $f(j)$ be the index of the column vector used as the jth step. Equation (6) becomes:

$$b = A_{f(1)} + \ldots + A_{f(t)} \tag{7}$$

Now, in order to rearrange, we just have to move b onto the same side as the column vectors and then divide it up to retain the small steps.

$$\begin{aligned} 0 &= A_{f(1)} + \ldots + A_{f(t)} - b \\ &= (A_{f(1)} - b/t) + \ldots + (A_{f(t)} - b/t) \end{aligned}$$

Both $A_{f(j)}$ and b/t are bounded by Δ, so their difference is bounded by 2Δ — still small!

Invoking Steinitz's lemma, for some particular permutation of f, we get that any partial sum with $j \leq t$ is bounded as follows:

$$\|A_{f(1)} + \ldots + A_{f(j)} - jb/t\|_\infty \leq 2\Delta m \tag{8}$$

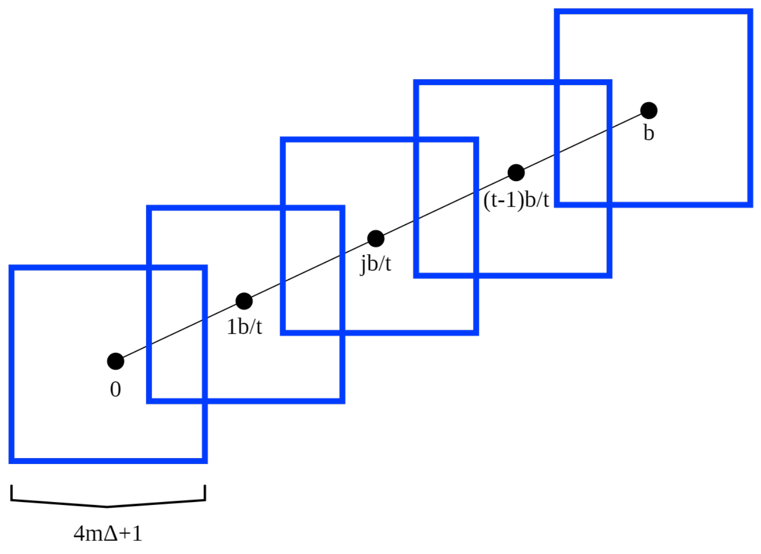

Figure 3: The space of possible paths from the origin to b

This bound limits the entire path to lie within a hypercube whose sides are distance $2\Delta m$ from the point jb/t. This means that each hypercube contains $4\Delta m + 1$ points per side and therefore $(4\Delta m + 1)^m$ points in total. The hypercube lives in m dimensions and there are t possible center points, so the set of all possible vertices the path could pass through is $|V| = t(4\Delta m + 1)^m$. Figure 3 shows this shape in 2 dimensions.

For each of these vertices there are at most n edges from that vertex, one for each column of A. Specifically there will be an edge going from vertex u to vertex v if $v - u$ is a column of A. This means that the graph we have constructed has at most $|E| = n|V| = nt(4\Delta m + 1)^m$ edges.

For feasibility, we just need to prove that a path exists. This can be done using breadth first search in $O(|E|) = O(nt(3\Delta m + 1)^m)$ time. To bound t, we can assume that the path does not contain zero columns, or sequences of columns that sum to zero as these are unnecessary to the final results. Therefore every step of the path must increase the ℓ_1 norm of our partial sum by at least 1, so $t \leq \|b\|_1$. This gives us the bound:

$$n \cdot O(m\Delta)^m \cdot \|b\|_1$$

3.3 Optimization

Optimization is not much different from feasibility. Rather than simply finding a path, we now need to find the path that optimizes $c^T x$. If we assign weights c_i to each edge corresponding edge, then the maximum weight path corresponds to the solution maximizing $c^T x$. Bellman-Ford can solve the problem with complexity $O(|V||E|) = O\left(nt^2(3\Delta m + 1)^{2m}\right) = n \cdot O\left((m\Delta)^{2m}\right) \cdot \|b\|_1^2$

Some care needs to be made with positive cycles. Just like negative cycles in the shortest path problem, these cycles can be looped around forever with every solution being better than the last. If these occur, they indicate that the solution is unbounded. Fortunately, Bellman Ford can detect these cycles and we can output "unbounded" if we find any.

4 Jansen and Rohwedder's Algorithm

In the Eisenbrand et al.'s algorithm, we learned that our solution could be considered as a path from the origin to b. Moreover, this path is constrained to lie within hypercubes with sidelengths $4\Delta m + 1$ centered around the the points jb/t for $j \leq t$ as we showed in figure 3. This series of center points lies exactly on the line from the origin to b, so one way of looking at the bound is that the path never strays far from this line.

Using this intuition Jansen and Rohwedder developed another, more efficient, solution to ILP using dynamic programming [5].

Specifically, they realized that if every point on the path is pretty close to the line from 0 to b, and the step sizes are small, then there must be some point that lies pretty close to the point $b/2$. If we guess the vector $b^{(1)}$ that is near to $b/2$, then we can solve the smaller problems $Ax = b^{(1)}$ and $Ax = b - b^{(1)}$ and then merge the solutions to get the full result. This sounds like dynamic programming!

In particular, the vectors we are "guessing" as halfway points can be described as the set of all $b^{(i)}$ such that $b^{(i)}$ is within distance $4m\Delta$ of a midpoint:

$$\left\| b^{(i)} - \frac{b}{2^i} \right\|_\infty \leq 4m\Delta \tag{9}$$

If we know the solution to $Ax = b^{(i+1)}$ for all possible $b^{(i+1)}$s satisfying (9), we can combine the results to get solutions to all $b^{(i)}$s, which are twice as far away the $b^{(i+1)}$s. Repeating this procedure until $i = 0$, we get the solution to $b = b^{(0)}$ which solves our problem.

First of all we will define the base case in section 4.1: how do we begin to get solutions to the problem for some large i? We also prove the dynamic programming step in section 4.2, which describes how we are able to compute solutions $i - 1$ from i. Then, we will discuss checking if the solution is unbounded in section 4.3 and conclude with an analysis of the runtime in section 4.4.

4.1 Base Case

In Papadimitriou's algorithm, we used a value B, to bound the ℓ_∞-norm of the optimal solution x^*. In this case we will be using a value of K to bound the ℓ_1 norm of the optimal solution. Specifically, $\|x^*\|_1 = 2^K$. As we divide b in half, the solution x will also be divided roughly in half. We can, in fact constraint the solution to do so:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax = b^{(i)} \\ & \|x\|_1 = 2^{K-i} \\ & x \in \mathbb{Z}^m_{\geq 0} \end{aligned}$$

With this constraint, when $i = K$ we have $\|x\|_1 = 1$; each such x is zero except for a single row with a 1. This means that these solutions are selecting single columns of A. We can easily iterate over each columns of A for each $b^{(K)}$ to find any matches. $b^{(K)}$s with matches will get value $c^T x$ and ones without will get value $-\infty$.

Now that we have solutions for a large $i = K$, we can use dynamic programming to iteratively find the solutions for smaller and smaller i until we reach the desired $i = 0$. Jansen et al. derive that $K = O(m \log(m\Delta) + \log(\|b\|_\infty))$. As an intuition for this bound, we recall again figure 3. This figure shows that x^* is constrained to live in one of at most t hypercubes, each with volume $(4m\Delta + 1)^m$. With $t \leq \|b\|_\infty + 1$, this means that the ℓ_1 norm is at most $\|x^*\| \leq O(m\Delta)^m(\|b\|_\infty + 1)$. Taking the log of this bound gives us K.

4.2 Induction

In order to apply dynamic programming, we need to prove that if we have solutions to $Ax = b^{(i+1)}$ for all $b^{(i+1)}$ satisfying equation (9), then we can construct solutions for all $b^{(i)}$ satisfying (9). To do that, we are actually going to go backwards. We are going to start with the path to some $b^{(i)}$ and prove that each half of that path is in the set of $b^{(i+1)}$s. Using the same notation we used in (7), we can write $b^{(i)}$ as columns of A:

$$b^{(i)} = A_{f(1)} + \ldots + A_{f(t)}$$

We claim that each half of the path is in the set of all $b^{(i+1)}$s. In other words, we are claiming that:

$$\left\| A_{f(1)} + \ldots + A_{f(t/2)} - \frac{b}{2^{i+1}} \right\|_\infty \le 4m\Delta \qquad \text{The left half is a } b^{(i+1)}$$
$$\left\| A_{f(t/2+1)} + \ldots + A_{f(t)} - \frac{b}{2^{i+1}} \right\|_\infty \le 4m\Delta \qquad \text{The right half is a } b^{(i+1)}$$

We'll focus on proving the left half, because the right half is similar. Without loss of generality, let's assume that the ordering of the columns, f, is done by Steinitz lemma. Under this assumption, equation (8) gives us this bound:

$$\left\| A_{f(1)} + \ldots + A_{f(t/2)} - \frac{b^{(i)}}{2} \right\|_\infty \le 2\Delta m$$

Now, with a little bit of algebra, we can do the proof:

$$\begin{aligned}
\left\| A_{f(1)} + \ldots + A_{f(t/2)} - \frac{b}{2^{i+1}} \right\|_\infty &= \left\| A_{f(1)} + \ldots + A_{f(t/2)} - \frac{b^{(i)}}{2} + \frac{b^{(i)}}{2} - \frac{b}{2^{i+1}} \right\|_\infty \\
&\le \left\| A_{f(1)} + \ldots + A_{f(t/2)} - \frac{b^{(i)}}{2} \right\|_\infty + \left\| \frac{b^{(i)}}{2} - \frac{b}{2^{i+1}} \right\|_\infty \\
&\le 2\Delta m + \frac{1}{2} \left\| b^{(i)} - \frac{b}{2^i} \right\|_\infty \\
&\le 4\Delta m
\end{aligned}$$

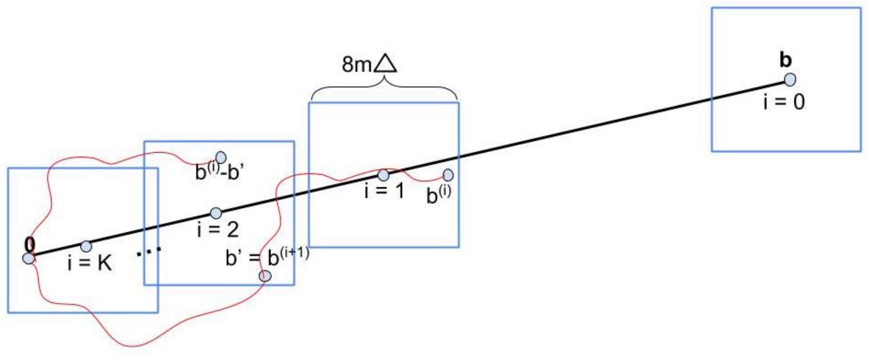

Figure 4: The path $b^{(i)}$ is the sum of paths $b' = b^{(i+1)}$ and $b^{(i)} - b'$.

4.3 Unbounded

So far, we have established how to find solutions to small subproblems, and how to combine those subproblems using dynamic programming to get a complete solution. However, what we have described so far will not be able to check if the solution is unbounded. Unlike in Eisenbrand et al.'s case, where checking if the solution was unbounded was simply built into Bellman-Ford, we will need to add some machinery here to do the check explicitly.

In particular, we are going to be solving a different ILP that is necessarily bounded. The solution to this

ILP will indicate whether the original ILP is unbounded. The new problem is:

$$\begin{aligned} \min \quad & \|x\|_1 && (10)\\ \text{s.t.} \quad & Ax = 0 && (11)\\ & c^T x > 0 && (12)\\ & x \in \mathbb{Z}^n_{\geq 0} && (13) \end{aligned}$$

If the ILP is feasible then the original ILP is unbounded, otherwise it is bounded. Essentially, what this ILP is doing is checking for positive cycles. The constraint $Ax = 0$ forces the steps to form a cycle that departs from the origin and returns to the origin. The constraint $c^T x > 0$ forces the cycle to be positive.

Now we need to make sure that we can solve this problem is itself bounded. Suppose that $t > (2\Delta m+1)^m$. Then realize that from our bound in equation (8) from the Steinitz lemma, the steps are bounded by

$$\|A_{f(1)} + \ldots + A_{f(j)}\|_\infty \leq 2\Delta m$$

This means that the partial sum of steps can take on at most $(2\Delta m + 1)^m$ unique values. If the number of steps is larger than $(2\Delta m + 1)^m$ there must be some value that has been repeated. This pins the total cycle together to make two cycles as shown in figure 5. Since $c^T x > 0$, then at least one of the cycles must be a positive cycle, which means there is a solution with a smaller $\|x\|_1$ that satisfies the ILP. This contradicts the minimality of $\|x\|_1$, so $t \leq (2m\Delta + 1)^m$. This means that we can solve this ILP with a bounded number of steps $t \leq (2m\Delta + 1)^m$. This is less than the number needed for the complete problem, so it will not affect the running time.

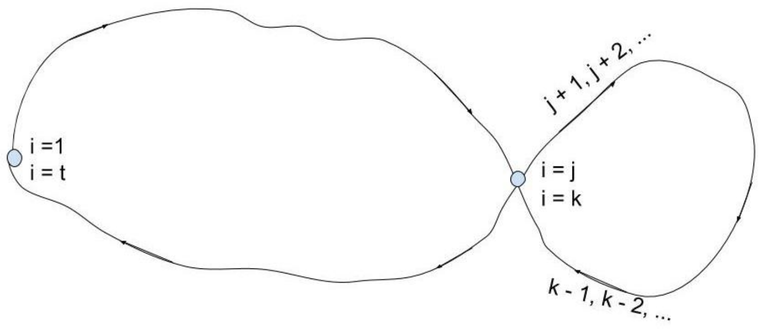

Figure 5: Positive cycle x^* decomposed into two cycles, where one of them is positive.

4.4 Runtime

So far, as we have constructed it, each layer of our dynamic programming table is limited by equation (9) to be contained in a hypercube with side lengths $8\Delta m + 1$. Because this is an m dimensional shape, its volume is $8\Delta m + 1 = O(\Delta m)^m$. Naively, updating each entry of the table requires iterating over all of the elements in the previous layer, requiring $(O(\Delta m)^m)^2 = O(\Delta m)^{2m}$ time.

In total, this table has $K + 1$ layers, which makes the total running time:

$$O(m\Delta)^{2m} \cdot (K + 1) = O(m\Delta)^{2m}(\log \Delta + \log \|b\|_\infty)$$

Jansen et al. propose several additional ways to reduce this running time further, removing the logarithmic factors. We will discuss them briefly here, along with their implications.

4.4.1 $(\max, +)$-convolution

The $(\max, +)$-convolution problem takes in two input sequences $r_1 \dots r_n$ and $s_1 \dots s_n$. Given these, the goal is to compute sequence $t_1 \dots t_n$ such that

$$t_k = \max_{i+j=k} r_i + s_j$$

Surprisingly, this simple little problem encapsulates the hardness of many other problems [2]. An algorithm exists with complexity $O(n^2/\log n)$ but it conjectured that no algorithm exists with complexity $O(n^{2-\epsilon})$ for any $\epsilon > 0$.

Bringing this back to Jansen et al.'s ILP algorithm, one of the areas for improvement is the dynamic programming step which iterates over every element of the previous layer for each element of the current layer in order to find the path with the maximum weight. With a little massaging, this problem is just like iterating over the sequences in the $(\max, +)$-convolution problem. Applying the algorithm, the complexity does not square with the size of the layer s, but instead has a slightly lower growth, $s^2/\log s$. This makes the complexity:

$$\frac{O(m\Delta)^{2m}}{2m \log O(m\Delta)}(K+1) = O(m\Delta)^{2m}(1 + \log(\|b\|_\infty)/\log(\Delta))$$

4.4.2 Proximity

This is a technique used by both Eisenbrand et al. and Jansen et al. Essentially, it uses the solution to the linear program to help bound the solution to the ILP even further. Eisenbrand et al. proved that the distance from the relaxed solution z^* to the actual solution x^* is

$$\|z^* - x^*\|_1 \leq m(2m\Delta + 1)^m$$

By using this bound in combination with the $(\max, +)$-convolution algorithm, the logarithmic factors vanish completely, giving an algorithm that runs in time:

$$O(m\Delta)^{2m} + \text{LP},$$

where LP is the time needed to compute the relaxed solution.

5 Unbounded Knapsack

As we discussed in the introduction, the unbounded knapsack problem with item values c_i and weights A_i can be written as the ILP:

$$\max \quad \sum_{i=1}^{n} c_i x_i \tag{14}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} A_i x_i = b \tag{15}$$

$$x \in \mathbb{Z}^n_{\geq 0} \tag{16}$$

In this problem, the A matrix is of shape $1 \times n$, meaning that $m = 1$. Having such a small m makes are otherwise exponential algorithms look much less scary! The best running time we have for ILP is $O(m\Delta)^{2m} +$ LP from Jansen et al. using proximity as mentioned in section 4.4.2. This makes the running time for unbounded knapsack $O(\Delta^2)$.

A classic dynamic programming algorithm for unbounded knapsack achieve $O(nC)$, where C is the sum of all item weights [1]. Another result achieves $O(n\Delta^2)$ [3]. So for $\Delta \ll C$ — which is reasonable if there are many items — the running time achieved by Jansen et al.'s ILP algorithm is the state of the art.

6 Conclusion

We have described the progress of integer linear programming algorithms that has flourished in the past couple of years. Eisenbrand et al.'s insight to use Steinitz's lemma made great improvements over the long-held bound established by Papadimitriou. Janset et al. built on Eisenbrand et al.'s work to achieve a running time as good as $O(m\Delta)^{2m} + \text{LP}$. Fomin et al. [4] have showed that if we assume the exponential time hypothesis, then Jansen et al.'s results are nearly optimal. In application, these results give state of the art bounds on problems with a limited number of constraints, such as the knapsack problem.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] Marek Cygan, Marcin Mucha, Karol Wegrzycki, and Michal Wlodarczyk. On problems equivalent to (min, +)-convolution. *CoRR*, abs/1702.07669, 2017.

[3] Friedrich Eisenbrand and Robert Weismantel. Proximity results and faster algorithms for integer programming using the steinitz lemma. *CoRR*, abs/1707.00481, 2017.

[4] Fedor V. Fomin, Fahad Panolan, M. S. Ramanujan, and Saket Saurabh. Fine-grained complexity of integer programming: The case of bounded branch-width and rank. *CoRR*, abs/1607.05342, 2016.

[5] Klaus Jansen and Lars Rohwedder. On integer programming and convolution. *CoRR*, abs/1803.04744, 2018.

[6] Christos H Papadimitriou. On the complexity of integer programming. *Journal of the ACM (JACM)*, 28(4):765–768, 1981.