Computing Disjoint Paths on Polytopes

David Avis$^1$ and Bohdan Kaluzny$^2$
School of Computer Science, McGill University,
Montreal, Canada
March 28th, 2005

Abstract

The Holt-Klee Condition states that there exist at least $d$ vertex-disjoint strictly monotone paths from the source to the sink of a polytopal digraph consisting of the set of vertices and arcs of a polytope $P$ directed by a linear objective function in general position. Studying these paths has the potential to provide new insight into a long standing problem, that of designing a polynomial-time simplex method, or proving none exists. To study these paths it would be useful to have a tool to compute them. Without explicitly computing the digraph we develop an algorithm to compute a maximum cardinality set of source to sink paths in a polytope, even in the presence of degeneracy. The algorithm uses a combination of networks flows, the simplex method, and reverse search. An implementation is available. Experimental results show that the algorithm excels when the input has little or no degeneracy, and is especially memory-efficient when the polytope has many vertices. For example, we computed 10 disjoint paths on the polar of the cyclic polytope of dimension 10 with 50 facets storing only 199,000 vertices while the polytope has 1,357,510 vertices. The median path length was 32 vertices. Further preliminary results show that the lengths of the disjoint paths are typically short.

La Condition Holt-Klee déclare que là existent au moins $d$ chemins distincts-de-sommet du entrée au sortie d’un graphe polytopal comprenant l’ensemble de sommets et le arêtes d’un polytope $P$ dirigé par une fonction linéaire en position générale. Étudier ces chemins a le potentiel de fournir la nouvelle perspicacit dans un problème de longue date, celle de concevoir une méthode simplexe de temps polynomial, ou en la preuve qu’aucune existe. Pour étudier ces chemins il serait utile d’avoir un outil pour les calculer. Sans calculer explicitement le graphe orienté nous développons un algorithme pour calculer un ensemble de cardinalité maximal de chemins distincts d’entrée au sortie dans un polytope, même si l’entrée est dégénérée. L’algorithme emploie une combinaison des flots à travers un réseau, du méthode simplexe, et de recherche inversé. Une exécution est disponible. Les résultats expérimentaux montre que l’algorithme excelle quand l’entrée a peu ou pas de dégénérésence, et est particulièrement mémoire-eficace quand le polytope a beaucoup de sommets. Par exemple, nous avons calculé 10 chemins distincts sur le polaire du polytope cyclique de la dimension 10 avec 50 facettes gardant seulement 199,000 sommets en mémoire tandis que le polytope a 1,357,510 sommets. La longueur de chemin médiane était 32 sommets. Encore d’autres résultats préliminaires montre que les longueurs des chemins distincts sont en général courtes.

$^1$avis@cs.mcgill.ca
$^2$bohdan.kaluzny@mcgill.ca
1 Introduction

A polyhedron is the intersection of \( n \) linear inequalities in \( d \) dimensions, and is referred to as a polytope if it is bounded. Throughout we assume that \( n > d \geq 2 \) and unless stated otherwise, we consider only polyhedra with full dimension. The study of paths on polyhedra has a long history, motivated by a desire to find a polynomial time version of the simplex method for linear programming, which is the problem of maximizing a linear function \( cx \) over a polyhedron. Polynomial time algorithms for linear programming, based on interior point methods, have since been discovered. However none of them are strongly polynomial, that is, run in time bounded by a polynomial in \( n \) and \( d \). Pivot based methods are still prime candidates for such an algorithm.

The vertices and edges of a \( d \)-polytope \( P \) form an undirected graph \( G(P) \). The diameter of \( G(P) \) is the smallest number \( \delta \) such that every two vertices in \( G(P) \) are connected by a path with at most \( \delta \) edges. The earliest, and most outstanding conjecture on the diameter of \( G(P) \) is the following.

Conjecture 1 (Hirsch Conjecture, [16] p. 84) Let \( P \) be a \( d \)-polytope with \( n \) facets, then the diameter of \( G(P) \) is at most \( n - d \).

A linear function \( cx \) in \( \mathbb{R}^d \) is in general position with respect to \( P \) if it attains different values at each vertex of \( P \). Such a function gives an orientation on the edges of \( G(P) \) allowing us to define an acyclic digraph \( D_c(P) \). This digraph has a unique source, or vertex minimizing \( cx \), and sink, or vertex maximizing \( cx \). The primal simplex method follows a path in \( D_c(P) \) from any given starting vertex to the sink. Even if true, the Hirsch Conjecture would not imply a bound on the length of such a monotone path. In fact, Todd [15] gave an example of 4-polytope with 8 facets for which every monotone path from a given vertex to the sink had at least 5 edges. A conjecture more directly related to paths taken by the simplex method is the following.

Conjecture 2 (Ziegler’s Strict Monotone Hirsch Conjecture, [16] p.86) Let \( P \) be a \( d \)-polytope with \( n \) facets and \( cx \) a linear function in general position, then there exists a path in \( D_c(P) \) from source to sink of length at most \( n - d \).

Holt and Klee proved that the Strict Monotone Hirsch Conjecture is true when \( d \leq 4 \). In doing so they unearthed the following interesting fact.

Theorem 1 (Holt-Klee Condition [9]) Let \( P \) be a \( d \)-polytope with \( n \) facets and \( cx \) a linear function in general position with respect to \( P \), then there exist \( d \) vertex-disjoint strict monotone paths from source to sink.

In this paper we give an algorithm for computing these paths for a polytope \( P \) given by a system of \( n \) linear inequalities in \( d \) variables. Given a directed graph, a maximum cardinality set of disjoint paths from source to sink can be found efficiently using network flow techniques, see Section 2 for details. However, in our case we are not given explicitly the digraph \( D_c(P) \). Computing \( D_c(P) \) would require
the enumeration of all vertices of $P$, a computationally difficult task, see Avis, Bremner and Seidel [3]. Worse, the storage requirement for $D_c(P)$ can be exponential in the input size. For example, the polar of a cyclic $d$-polytope with $n$ facets has $\Theta(n^{\frac{d}{2}+1})$ vertices. Complete vertex enumeration and storage would be required even if just a few disjoint paths are required. Our algorithm avoids computing $D_c(P)$ explicitly, using an oracle based on the simplex method to provide edges.

A further complication is caused by degeneracy. A vertex of $P$ is called simple if it is contained in exactly $d$ facets of $P$, and degenerate otherwise. A polytope is simple if all of its vertices are simple, and degenerate otherwise. A set of $d$ affinely independent facets containing a vertex is called a cobasis of the vertex. The complementary set of facets is called a basis. The set of all bases of $P$ defines an undirected graph, called the basis graph $B(P)$: the vertices of $B(P)$ are bases of $P$, and the edges of $B(P)$ are defined by the pivot operation, see Section 3 for details. We will avoid using the term “vertex of $B(P)$” to avoid confusion with the the term “vertex of $P$”. Instead we will refer to the “bases” of $B(P)$. For a simple polytope $P$, the graphs $B(P)$ and $G(P)$ are identical. However, the graph $B(P)$ can be much larger than the graph $G(P)$, since a highly degenerate vertex may be representable by an exponential number of bases. The simplex method follows paths in $B(P)$.

Let $cx$ be in general position with respect to $P$. A path in $B(P)$ is said to be monotone with respect to $c$, if $cx$ is nondecreasing when applied to the vertices of $P$ corresponding to consecutive bases in the path. In the case of nondegeneracy, $cx$ increases strictly along the path. In the presence of degeneracy, a degenerate vertex appears as one or more consecutive bases on the path. The vertices of $P$ as visited by a monotone path in $B(P)$ induce a strictly increasing path in $D_c(P)$. Note however, that to find vertex-disjoint monotone paths in $D_c(P)$ it is not sufficient to find basis disjoint monotone paths in $B(P)$: in the presence of degeneracy basis disjoint paths are not necessarily vertex-disjoint.

In this paper we present an algorithm and an implementation for finding $d$ vertex independent monotone paths in $D_c(P)$ from a basis of the source to a basis of the sink of $P$. The ingredients of the algorithm are:

- the simplex method to determine edges of the graph $B(P)$;
- a network flow algorithm to determine disjoint paths in $B(P)$;
- reverse search to handle degenerate vertices in a space efficient manner.

In Section 2 and 3 we give the details of these ingredients. Combining these we then present our algorithm in Section 4. In Section 5 we discuss complexity, implementation, and some experimental results. In the implementation, we will not in fact require that $cx$ is in general position with respect to $P$, although for simplicity this will be assumed in describing the algorithm in Section 4.

2 Disjoint Paths in a Digraph

Let $D$ be a directed graph with two specified vertices $s$ and $t$. An algorithm for finding the maximum number of edge-disjoint $s-t$ paths in $D$ is quite simple and can be found in most graph theory textbooks such as [14]. It is a specialization of the maximum flow algorithm of Ford and Fulkerson [8]: find a directed path from $s$ to $t$ in $D$, reverse the direction of the edges along this path, and repeat
this process until no further path is found.

For a directed path $\overrightarrow{P}$ in a digraph $D$, let $D \leftarrow \overrightarrow{P}$ be the digraph arising from $D$ by reversing the orientation of each arc in $\overrightarrow{P}$.

**Algorithm 1 (Edge-Disjoint Paths)** Determine $D_0, D_1, ...$ as follows:

Set $D_0 := D$.

**Find Path($D_i$)**: Find an $s - t$ path $\overrightarrow{P}$ in $D_i$

If $\overrightarrow{P}$ is found

then **Reverse Path($D_i, \overrightarrow{P}$)**: set $D_{i+1} := D_i \leftarrow \overrightarrow{P}$.

Otherwise stop.

**Proposition 1** The set $R_i$ of arcs of $D$ that are reversed in $D_i$ form $i$ edge-disjoint $s - t$ paths.

**Proposition 2** The edge-disjoint paths algorithm finds a maximum collection of edge-disjoint $s - t$ paths.

See [14, p.135] for proofs. Figure 1 shows the algorithm at work. The dashed lines in Figures 1 (a)-(c) show the new path found at each iteration. The solid edges in Figure 1(d) indicate three edge-disjoint paths from source to sink, with edges reversed.

A simple modification of the algorithm allows us to find vertex-disjoint paths. Given a digraph $D$, create $D'$ by replacing every vertex $v$ in $D$ by two vertices $v', v''$ and an arc $v'v''$. Replace each arc $uv \in D$ by $uv'$, and $vw \in D$ by $v''w$. Now edge-disjoint paths in $D'$ are easily mapped to vertex-disjoint paths in $D$. Instead of creating new vertices and edges, we can modify the algorithm to simply mark a vertex $v$ in $D$ if it is used in a path. In **Find Path($D_i$)** we check whether a vertex $v$ is marked, signifying that the artificial edge $v'v'' \in D'$ is reversed. At a vertex $v$, let $u$ be the predecessor of $v$, then only in the following cases is $vw$ an admissible edge in $D_i$. (In the corresponding figures, reversed edges are dotted, a black-filled vertex is marked, white-filled vertex unmarked, and grey-filled vertex could be either marked or unmarked.)

**Case 1:** $v$ is not marked, $uv$ and $vw$ are not reversed

**Case 2:** $v$ is marked, $vu$ and $wv$ are reversed
Case 3: $v$ is marked, $uv$ is not reversed, and $wv$ is reversed

Case 4: $v$ is marked, $vu$ is reversed and $vw$ is not reversed

Theorem 2 A collection of $k$ internally vertex-disjoint $s-t$ paths can be found in $O(k|E|)$ time.

Proof. A path in a graph can be found in $O(|E|)$ time with a graph search algorithm like depth-first-search (dfs). □

Of course Theorem 2 assumes that we have easy access to the vertices and edges of the graph. Unfortunately the polytopal digraph $D_c(P)$ is not readily available from our input. Instead we define the function $\text{Find Path}(D_i)$ of Algorithm 1 by using an adjacency oracle based on the memory-less local search operation of the simplex method, as described in the next section. Then the idea is simple. We use the simplex method to compute the first path $\vec{P}_1$. We store each vertex of $D_c(P)$ that we encounter on the path $\vec{P}_1$ in a data structure named $R$. Thus we have found our first path and reversed its edges, abstractly setting $D_1 := D_c(P)$ and $D_2 := D_1 ← \vec{P}_1$. For successive iterations we initialize a data structure $S$ to be used for marking nodes in a dfs of the vertices. We order the edges incident to each vertex so that given a vertex $v$ and the last edge followed from $v$, we can define an adjacency oracle to return the next vertex adjacent to $v$. To find the next path, starting from the source vertex, we repeat the following dfs process until the sink is found: if $v ∉ S$ add the current vertex $v$ to $S$, query the adjacency oracle and either find a vertex $w ∉ S$ where $vw$ is the next admissible edge from $v$ in $D_i$, or backtrack to the dfs predecessor of $v$. The next section describes the adjacency oracle.

3 Finding Edges: Simplex Method and Reverse Search

In this section, we assume basic familiarity with the Simplex Method, and refer the reader to Chvátal [5] for further background. Let $v$ be any vertex of a $d$-polytope $P$, let $\Delta$ be an upper bound on the maximum degree of any vertex in $G(P)$, and let $cx$ be in general position with respect to $P$. In this section we describe an oracle $\text{Adj}(v,j), j = 1, ..., \Delta$, with the following properties:
- $\text{Adj}(v,j)$ is either empty or a vertex $w$ of $P$ such that $vw$ is an edge in $G(P)$.
- as $j$ ranges over all possible values, each vertex $w$ adjacent to $v$ appears exactly once.

The polytope $P$ is given as a system of inequalities:

$$\sum_{j=1}^{d} a_{ij}x_j \leq b_i \quad \text{for } i = 1...n. \quad (1)$$

A linear program is formed by maximizing the linear objective function

$$z = cx = \sum_{j=1}^{d} c_jx_j \quad (2)$$
over $P$. By adding slack variables $x_{d+1}, \ldots, x_{n+d}$, we can write this linear program in dictionary form:

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{for } i \in B$$

$$z = z' + \sum_{j \in N} c'_j x_j,$$

where initially

$$B = \{1, 2, \ldots, d\}, N = \{d + 1, d + 2, \ldots, n + d\}, z' = 0, a'_{ij} = a_{ij}, b'_i = b_i, c'_j = c_j \text{ for all } i, j. \quad (4)$$

The variables on the left-hand side of (3), indexed by $B$, are called basic and the right-hand side variables, indexed by $N$, cobasic. Pivoting is the operation of exchanging a cobasic variable for a basic variable. We evaluate a dictionary by setting the cobasic variables to zero to get a basic solution. When all $n$ slack variables are nonnegative then it is a basic feasible solution, and defines to a vertex of $P$. Conversely, for each vertex of $P$ there correspond one or more basic feasible solutions, and the correspondence is one to one if $P$ is simple. The simplex method performs pivots that preserve basic feasibility, and this is achieved by a ratio test to choose the variable to leave the basis. We will only consider feasible pivots, and they are used to define the oracle $Adj(v, j)$. We first discuss the case where $P$ is simple.

If $P$ is simple then each basic feasible solution corresponds to a vertex $v$ of $P$ with a unique basis $B$. The ratio test is defined as follows:

$$\text{ratio}(j) = \arg\min \left\{ \frac{b'_i}{a'_{ij}} : i \in B, a'_{ij} > 0 \right\}, \quad (5)$$

where argmin returns the index $i$ minimizing the given ratio. The corresponding pivot replaces the cobasic variable $x_j$ by the basic variable $x_{\text{ratio}(j)}$, and vice versa. Since $P$ is simple, we must have $b_i > 0$ and the pivot yields a dictionary representing a neighbour $w$ of $v$. If $c'_j > 0$ then $vw$ is an arc in $D_c(P)$, otherwise if $c'_j < 0$ then $wv$ is an arc in $D_c(P)$. We may set $\Delta = n$ and define the oracle by:

$$Adj(v, j) = w \quad \text{if} \quad j \in N$$

$$= \emptyset \quad \text{otherwise.} \quad (6)$$

The degenerate case is more challenging. We begin with an example to illustrate the difficulties encountered. Consider the 3-polytope $P$ defined by

$$-2x_1 + 2x_2 + x_3 \leq 2$$

$$x_2 + x_3 \leq 3$$

$$2x_1 + x_2 + 3x_3 \leq 9$$

$$x_1 - 2x_2 + \frac{3}{2} x_3 \leq 2$$

$$-2x_1 - 4x_2 + x_3 \leq -4$$

$$-x_3 \leq 0.$$
$P$ is a pyramid over a 5-gon with six vertices: $a : (0,1,0), b : (2,3,0), c : (3,3,0), d : (4,1,0), e : (2,0,0), f : (1,1,2)$. Vertex $f$ is degenerate, since it is contained in five facets: the first five inequalities in (7) are satisfied as equations. If we choose $c = (-1,0,0)$ the linear function $z = cx = -x_1$ defines the polytopal digraph $D_c(G)$ shown in Figure 2.

![Figure 2](image)

Vertex $f$ can be represented by the dictionary with cobasis $\{6,7,8\}$:

\begin{align*}
    x_1 &= 1 + \frac{1}{5}x_6 - \frac{13}{20}x_7 + \frac{3}{8}x_8 \\
    x_2 &= 1 - \frac{1}{5}x_6 + \frac{2}{5}x_7 \\
    x_3 &= 2 - \frac{2}{5}x_6 + \frac{3}{10}x_7 - \frac{1}{4}x_8 \\
    x_4 &= 0 + \frac{6}{5}x_6 - \frac{12}{5}x_7 + x_8 \\
    x_5 &= 0 + \frac{3}{5}x_6 - \frac{7}{10}x_7 + \frac{1}{4}x_8 \\
    x_9 &= 2 - \frac{2}{5}x_6 + \frac{3}{10}x_7 - \frac{1}{4}x_8 \\
    z &= -1 - \frac{1}{5}x_6 + \frac{13}{20}x_7 - \frac{3}{8}x_8.
\end{align*}

Consider for example a pivot on the cobasic variable $x_7$. The minimum ratio computed by (5) is zero and achieved by the basic variables $x_4$ and $x_5$. The corresponding pivots are degenerate pivots since they lead to other bases representing $f$. Observe that no pivot from (8) yields the edge $af$ to vertex $a$ whose cobasis is $\{4,8,9\}$. The basis graph $B(P)$ is shown in Figure 3. (In all figures we label vertices by the corresponding cobasic indices, since there are fewer of these than there are basic indices.) Note that vertex $f$, which is contained on 5 facets, is represented by $\binom{5}{2} = 10$ bases.
The adjacency oracle must return all the edges of $G(P)$ for each vertex $v$. This is known as the neighborhood problem. When $v$ is degenerate, we could achieve this by enumerating all the bases of $v$. The edges of $v$ could then be extracted by considering nondegenerate pivots from each basis of $v$. Enumerating the bases of a degenerate vertex $v$ can be a daunting task: if $v$ is contained in $k \geq d$ facets, then $v$ can be represented by up to ${k \choose d}$ bases. Fortunately a subset of these bases, known as *lex-positive* bases, are sufficient. The lex-positive bases are in one to one correspondence with the vertices of a simple polytope obtained by a perturbation of the inequalities defining $P$. By McMullen’s Upper Bound Theorem [12] this limits the number of lex-positive bases to at most $\Theta(k^{\lfloor d/2 \rfloor})$. Whilst a big improvement, this bound is still exponential. The subgraph of $B(P)$ containing just the lex-positive bases is given in Figure 4. Fortunately, determining all the lex-positive bases can be achieved in a space efficient manner by making use of the reverse search method developed by Avis and Fukuda [4]. Its application to finding all the lex-positive bases of a polyhedron is described in [2], and is the basis of the *lrs* code for generating all vertices of a polyhedron. We give just a brief summary here, and then explain how to specialize it for our purposes.

Every vertex of a $d$-polytope $P$ can be represented by a lex-positive basis. In fact the *lex-min basis* of a vertex, defined as the lexicographically smallest subset of indices that represent that vertex, is lex-positive ([2], Proposition 5.2.) A *lexicographic simplex pivot* preserves the lex-positive property of the bases by employing a *lexicographic ratio test* which replaces the test described in (5). Note that in case of degeneracy, the test (5) may not give a unique index $i \in B$, but this is always guaranteed by the lexicographic ratio test. The *lexicographic pivot rule* applied to a feasible dictionary selects the smallest index $j$ such that $c_{ij} \geq 0$ to determine the entering index $j$, and uses the lexicographic ratio test to find the leaving index $i$. If there is no such index $j$, the dictionary is optimum. There is unique optimum dictionary with lex-positive basis, and this is the lex-min basis for the optimum basis ([2], Proposition 4.3). The lexicographic pivot rule therefore defines a spanning tree on the set of lex-positive bases of $P$, rooted at the lex-min basis of the optimal vertex. This tree, which contains at least one basis for each vertex of $P$, can be transversed without additional storage using the reverse search method. We adapt this method to determine all lex-positive bases of any degenerate vertex $v$.  

---

Figure 3: Basis graph $B(P)$  
Figure 4: Lex-positive subgraph
Algorithm 2 (Lexicographic Reverse Search on a Degenerate Vertex) Designate the lex-min basis of \( v \) as the root of a spanning tree \( T \) on the lex-positive bases for \( v \), by creating an objective function that is optimal at this basis. There exists a lexicographic simplex pivot path from every lex-positive basis of \( v \) to the (optimal) lex-min basis. The collection of these paths define the enumeration tree \( T \) of \( v \). Reverse search traces out \( T \) in a depth first manner without using additional storage:

- To find children of a node \( t \in T \) reverse the lexicographic pivot rule: \( t \) is a parent of \( r \) if the degenerate pivot from \( r \) to \( t \) satisfies the lexicographic pivot rule.
- The parent of a node \( t \) in \( T \) is found by a single pivot following the lexicographic pivot rule.

For our example, we show in Figure 5 a reverse search tree for \( f \), as a subtree of the full reverse search tree for \( P \), rooted at \( f \). The dotted edges show the degenerate pivots performed by Algorithm 2. Note the tree is rooted at the lex-max cobasis for \( f \), \( \{6, 7, 8\} \), for which the corresponding basis is lex-min.

![Figure 5: Reverse search tree of lex-positive bases rooted at (6, 7, 8)](image)

We can now fully define the adjacency oracle \( \text{Adj}(v, j) \). We are given a dictionary with basis \( B \) representing a vertex \( v \), and index \( i \) of the last cobasic variable pivoted on in \( B \). The adjacency oracle attempts to find the next admissible edge from \( B \) by incrementing \( i \) until either \( i > d \), or the pivot on this column of the dictionary is nondegenerate. In the latter case, this nondegenerate pivot defines a vertex \( w \) such that \( vw \) is an edge of \( G(P) \). The oracle returns \( \text{Adj}(v, j) = w \) and \( j \) is incremented. If \( i > d \), then the oracle pivots to the next node in the reverse search tree of \( v \), resets \( i = 0 \), and repeats this process until the next edge is found or all lex-positive bases for \( v \) have been considered.

4 Description and Implementation of the Algorithm

In this section we give a more complete description of the algorithm and its implementation. To store and identify a vertex we use the cobasis of its unique lex-min basis as its representative. The top level of the algorithm uses depth first search, \( \text{dfs} \), to find paths in \( D_c(P) \) from source to sink, as described in Section 2. We require two basic data structures.
• **S**: An AVL tree storing the vertices of \( D_c(P) \) that have been visited by \( dfs \). A node of the \( dfs \) tree, representing a vertex \( v \) of \( P \), will consist of the cobasis of the lex-min basis of the vertex as the search key. Additionally we store information for backtracking, namely a pointer to the to its \( dfs \) predecessor, and in the case of a degenerate vertex, an encoding of the cobasis, pivot indices, artificial objective function, and depth of the last node considered in the reverse search tree for \( v \). We also store pointers to the left son, right son, and balance factor of the AVL tree.

• **R**: An AVL tree storing the vertices and edges of \( D_c(P) \) that have been reversed. A node of the tree, representing a vertex of \( P \), will consist of the cobasis of the lex-min basis of the vertex as the search key. Additionally we will have two pointers to the two adjacent vertices stored in \( R \) (reversed edges associated with this vertex), as well as to the left son, right son, and balance factor of the AVL tree.

By Proposition 1, the \( k \) disjoint paths found at the \( k^{th} \) iteration are represented by vertices stored in \( R \).

**Algorithm 3 (k Vertex-Disjoint Monotone Paths)**

**Input**: \( n \) inequalities in \( d \) variables representing a full dimensional \( d \)-polytope \( P \), a linear function \( cx \) in general position with respect to \( P \), and an integer \( k \).

**Output**: \( k \) vertex-disjoint monotone paths in \( D_c(P) \) from the source to sink.

1. Initialize data structure \( R \). Find an initial path \( \overrightarrow{P} \) using the lexicographic simplex method and store the vertices of \( \overrightarrow{P} \) in \( R \) \( (D_1 := D_0 \leftarrow \overrightarrow{P}) \). Iteration := 1.

2. Initialize \( S := \emptyset \).

3. Find a new path \( \overrightarrow{P} \) : /*FindPath\((D_i)\)*/

   (3a) Set \( w := \text{lex-min basis of source}, b := w \) and \( i := 0, \text{pred} := \text{null} \). mark_vertex\((w,b,i,\text{pred})\).

   (3b) Set \( v := w \). If \( v \) is the sink then go to (4), otherwise set \( b := v \) and \( i := 0 \).

   (3c) \( w := \text{next_edge_oracle} (v,b,i) \). If \( w = \emptyset \), set \( w := \text{backtrack} (v) \) and goto (3b). Else if \( w \notin S \), and \( vw \) is admissible then update\((v,b,i)\), set \( b := w, i := 0, \text{mark_vertex} (w,b,i,v) \), and goto (3b).

4. Reverse each vertex \( p \) of \( \overrightarrow{P} \) : /*ReversePath\((D_i, \overrightarrow{P})\)*/

   (4a) If \( p \notin R \), add \( p \) and the edges adjacent to \( p \) in \( \overrightarrow{P} \) to \( \overrightarrow{P} \) incident to \( p \) to \( R \), otherwise remove \( p \) from \( R \).

   (4b) Iteration++. If Iteration == \( k \), then go to (5), otherwise go to (2).

5. Output the vertices and edges in \( R \) (up to \( k \) paths). /*End*/
next_edge_oracle(v, b, i): /*The Adjacency Oracle*/

(a) Increment i. If i > d goto (c).

(b) Perform a lexicographic pivot on the $i^{th}$ cobasic variable. If the pivot is nondegenerate, return the lex-min basis $w$ of the new vertex. Otherwise goto (a).

(c) If $v$ is degenerate, pivot to the next basis $b'$ in reverse search tree (Algorithm 2). If the reverse search tree is exhausted, or $v$ is nondegenerate then return $w := \emptyset$. Otherwise set $b := b', i := 0$ and goto (a).

mark_vertex(v, b, i, w): If $v \notin S$ then add node $s := \{v, b, i, w\}$ to $S$.

update(v, b, i): Update $b$ and $i$ and of node storing $v$ in $S$.

backtrack(v): If $v ==$ source the goto (5). Otherwise backtrack to predecessor $w$ in dfs search, load the basis $b$ of $w$ and index $i$ last used.

The output of the algorithm gives a set of strict monotone vertex-disjoint paths from source to sink in the polytopal digraph $D_c(P)$. If $P$ is simple, this is also a set of disjoint strict monotone paths in the basis graph $B(P)$, each edge of which is a pivot. If $P$ is degenerate, we can also efficiently produce a set of disjoint monotone paths in $B(P)$. For each degenerate vertex $v$ contained in a path, we may have two bases: $B_{in}$ representing the dictionary for $v$ when it is entered, and $B_{out}$ when it is left. These two bases can be easily joined by a path in $B(P)$. We make pivots interchanging $x_i$ and $x_j$, where $i \in B_{out} - B_{in}$ and $j \in B_{in} - B_{out}$. Since these pivots are degenerate, the set of disjoint paths formed in $B(P)$ are monotone.

We illustrate the algorithm on our previous example (7). Let the letters associated with each vertex represent the lex-min basis at that vertex. Assume that we have already found two source to sink paths, and that the data structures are set as $R = \{(e : (a, d), b : (c, f), c : (b, d), f : (a, b))\}$ and $S = \emptyset$. Starting from the source vertex $d$ represented by cobasis $\{6, 7, 9\}$ we mark vertex $d$, $S = \{d : (pred = \emptyset, leave = \emptyset, depth = \emptyset, obj = \emptyset)\}$.

$$
x_1 = 4 - \frac{2}{3}x_6 - \frac{1}{3}x_7 - \frac{3}{2}x_9 \\
x_2 = 1 - \frac{1}{3}x_6 + \frac{5}{3}x_7 \\
x_3 = x_9 \\
$$

$$
x_4 = 8 - \frac{2}{3}x_6 - \frac{6}{5}x_7 - 4x_9 \\
x_5 = 2 + \frac{1}{3}x_6 - \frac{7}{5}x_7 - x_9 \\
x_8 = 8 - \frac{2}{3}x_6 + \frac{3}{5}x_7 - 4x_9 \\
z = -4 + \frac{3}{5}x_6 + \frac{5}{3}x_7 + \frac{2}{5}x_9 \\
$$

The adjacency oracle considers the first nondegenerate pivot from $d$, pivoting on $x_6$ and $x_8$, to vertex $e : \{7, 8, 9\}$. $(d, e)$ is an arc in $D_c(P)$, as $c_6 = \frac{2}{3}$ is positive in the $z$-row, but $(d, e) \in R$ and hence not admissible. The oracle examines the next edge, the pivot on $x_7$ and $x_5$, to vertex $c$ which is also
inadmissible as \((d, c)\) is reversed. The oracle’s last option, pivot on \(x_9\) and \(x_8\), is a nondegenerate pivot to vertex \(f\). \(f\) is a degenerate vertex and cobasis \(\{6, 7, 8\}\) represents the lex-min basis for \(f\). \((d, f)\) is an arc in \(D_c(P)\) that is not reversed. We mark \(f\), \(S = \{d, f : (\text{pred} = d, \text{leave} = \{6, 7, 8\}, \text{depth} = 0, \text{obj} = 0)\}\).

\[
\begin{align*}
x_1 &= 1 + \frac{13}{20}x_6 - \frac{13}{20}x_7 + \frac{3}{8}x_8 & x_1 &= 1 + \frac{13}{20}x_5 - \frac{5}{14}x_6 + \frac{1}{7}x_8 & x_1 &= 1 + \frac{5}{12}x_4 - \frac{7}{25}x_5 + \frac{1}{12}x_8 \\
x_2 &= 1 - \frac{3}{8}x_6 + \frac{3}{8}x_7 & x_2 &= 1 - \frac{3}{8}x_5 + \frac{3}{8}x_6 + \frac{1}{7}x_8 & x_2 &= 1 - \frac{9}{6}x_4 + \frac{1}{6}x_8 \\
x_3 &= 2 - \frac{1}{2}x_6 + \frac{1}{2}x_7 - \frac{1}{4}x_8 & x_3 &= 2 - \frac{3}{8}x_5 - \frac{1}{7}x_6 - \frac{1}{7}x_8 & x_3 &= 2 + \frac{3}{6}x_4 - x_5 - \frac{1}{6}x_8 \\
x_4 &= 0 + \frac{3}{8}x_6 - \frac{12}{20}x_7 + \frac{3}{8}x_8 & x_4 &= 0 + \frac{4}{2}x_5 - \frac{3}{2}x_6 + \frac{1}{7}x_8 & x_4 &= 0 - \frac{7}{6}x_4 + 4x_5 + \frac{1}{6}x_8 \\
x_5 &= 0 + \frac{3}{8}x_6 - \frac{12}{20}x_7 + \frac{3}{8}x_8 & x_5 &= 0 - \frac{10}{9}x_5 + \frac{6}{5}x_6 + \frac{5}{12}x_8 & x_5 &= 0 - x_4 + 2x_5 + \frac{1}{6}x_8 \\
x_6 &= 2 - \frac{1}{2}x_6 + \frac{1}{2}x_7 - \frac{1}{4}x_8 & x_6 &= 2 - \frac{3}{8}x_5 - \frac{1}{7}x_6 - \frac{1}{7}x_8 & x_6 &= 2 + \frac{3}{6}x_4 - x_5 - \frac{1}{6}x_8 \\
z &= -1 - \frac{3}{8}x_6 + \frac{13}{20}x_7 - \frac{3}{8}x_8 & z = -1 - \frac{13}{14}x_5 + \frac{5}{14}x_6 - \frac{1}{7}x_8 & z &= -1 - \frac{5}{12}x_4 + \frac{2}{5}x_5 - \frac{17}{12}x_8 \\
z_{\text{r.s.}} &= 0 - x_6 - x_7 - x_8, & z_{\text{r.s.}} &= \frac{9}{14}x_5 - \frac{13}{20}x_6 - \frac{10}{14}x_8 & z_{\text{r.s.}} &= \frac{13}{6}x_4 - 6x_5 - \frac{5}{3}x_8
\end{align*}
\]

We now query the oracle for an outgoing edge from \(f\). The first nondegenerate pivot from \(\{6, 7, 8\}\) is \(x_6\) for \(x_9\) to vertex \(e\). Since \(e'_6 = -\frac{1}{3}\) is negative and arc \((e, f)\) is not reversed, this pivot is inadmissible. The nondegenerate pivot \(x_6\) for \(x_9\), to vertex \(d\) is inadmissible as \(d \in S\). As no more nondegenerate pivots are possible from \(\{6, 7, 8\}\) the oracle initiates a reverse search on the lex-positive bases of \(f\), setting the lex-max cobasis \(\{6, 7, 8\}\) as the root of a reverse search tree (refer to Figure 5). We construct an artificial objective function \(z_{\text{r.s.}}\), and also carry along the original objective function, although this is not used in pivot selection during this phase. Following Algorithm 2, the next cobasis attained is \(\{5, 6, 8\}\). We re-instate the \(LP\)’s objective function.

From this cobasis, the only nondegenerate pivot finds vertex \(c\) which is inadmissible since arc \((c, f)\) of \(D_c(P)\) is not reversed. The oracle restarts the reverse search from \(\{5, 6, 8\}\) and finds the next cobasis \(\{4, 5, 8\}\) of degenerate vertex \(f\). The first nondegenerate pivot from \(\{4, 5, 8\}\) finds sink vertex \(a\), however we check \(R\) and find that the arc \((f, a)\) in \(D_c(P)\) is reversed. The other nondegenerate pivot from \(\{4, 5, 8\}\) yields an admissible edge to vertex \(b\). We update \(S = \{d, f : (\text{pred} = d, \text{leave} = \{4, 5, 8\}, \text{depth} = 2, \text{obj} = z_{\text{r.s.}})\}\). The algorithm continues and finds an edge to vertex \(a\). \(S = \{d, f : (\text{pred} = d), b : (\text{pred} = f), a : (\text{pred} = b)\}\). Following the predecessor links in \(S\) we reverse the edges along the path taken, storing only the arcs not in \(D_c(P)\). \(R = \{e : (a, d), b : (a, c), c : (b, d), f : (a, d)\}\).

5 Computational Results & Complexity

We have implemented the \(k\) vertex-disjoint monotone paths algorithm as the program \textit{disjointlp}, using library functions of \textit{lrs} [17] for lexicographic pivoting and rational arithmetic. All computations are done exactly using extended precision arithmetic. The program and a user guide are available [18]. The input files are in standard \textit{polyhedra format} ([17]), and the program outputs \(k\) vertex-disjoint monotone paths, each as a sequence of vertices (represented by the cobasis of the lex-min basis) from source to sink. For example, the Klee-Minty cube \(LP\) in 3D has the following input and output:
Input:

```
H-representation
begin
6 4 rational
0 1 0 0
1 -1 0 0
0 -1 3 0
3 -1 -3 0
0 0 -1 3
3 0 -1 -3
end
disjoint 3
maximize 0 0 0 1
```

Output:

Disjoint Paths Computed:
Path 3: [1 3 5] -- [1 3 6]

Disjoint path statistics:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longest path length</td>
<td>4</td>
</tr>
<tr>
<td>Shortest path length</td>
<td>2</td>
</tr>
<tr>
<td># of path vertices</td>
<td>6</td>
</tr>
<tr>
<td>Median path length</td>
<td>4</td>
</tr>
<tr>
<td>Max # reversed nodes</td>
<td>6</td>
</tr>
<tr>
<td>Avg # reversed nodes</td>
<td>4.67</td>
</tr>
<tr>
<td>Max # marked nodes</td>
<td>7</td>
</tr>
<tr>
<td>Avg # marked nodes</td>
<td>2.67</td>
</tr>
<tr>
<td>Total # of pivots</td>
<td>39</td>
</tr>
</tbody>
</table>

The only assumption is that input inequalities define a full dimensional polytope $P$. The objective function does not need to be in general postion with respect to $P$, as dual degeneracy is broken by lexicographic ordering of the cobasic indices. For a dual degenerate edge, $vw$ is an arc if lex-min basis of $v$ is lexicographically smaller than the lex-min basis of $w$.

### 5.1 Computational Results

The implementation was used to compute a set of disjoint paths of maximum cardinality for various LP’s both nondegenerate and degenerate. For each problem we list the number of vertices and lex-positive bases that the polytope has, the amount of additional memory used on top of storing the vertices of the disjoint paths, the path lengths (# vertices), the total number of pivots operations from the time the input is loaded, and the number $k$ of disjoint paths computed. Apart from the Klee-Minty LP’s, we generated random objective functions by picking coefficients from the set $\{-1000, ..., 1000\}$.
The worst case for our algorithm is illustrated by the extremely degenerate examples in Table 2. The polytopal graphs for all are highly connected (e.g. the cut polytope’s graph is a complete graph), so the problem of finding vertex-disjoint paths is in fact trivial. There are relatively few vertices, but these are extremely degenerate, making basis enumeration impractical.

### Table 1: Nondegenerate examples

| name        | n   | d   | k   | |V| | memory used | path lengths | # pivots |
|-------------|-----|-----|-----|------|------|-------------|-------------|---------|
|             |     |     |     |     |     | max, avg    | max, min, med |         |
| Hypercube   | 10  | 5   | 5   | 32   | 16,8.2 | 6,6,6       |             | 182     |
| Hypercube   | 16  | 8   | 8   | 256  | 36,20.0| 9,9,9       |             | 899     |
| Klee-Minty LP | 16 | 8   | 8   | 256  | 254,161.1| 72,2,23     |             | 6078    |
| Klee-Minty LP | 18 | 9   | 9   | 512  | 510,352.7| 114,2,48    |             | 16225   |
| Klee-Minty LP | 20 | 10  | 10  | 1024 | 1022,730.4| 282,2,67    |             | 37736   |
| Klee-Minty LP | 30 | 15  | 15  | 32768| 32766,23953.5| 4310,2,1284|             | 1988459 |
| Polar Cyclic | 20  | 5   | 5   | 272  | 121,67.7| 17,10,12    |             | 1691    |
| Polar Cyclic | 50  | 10  | 10  | 1357510| 199000,21479.9| 70,28,32   |             | 1962605 |
| Polar Cyclic | 60  | 10  | 10  | 3795012| 709380,123209.0| 171,29,150.5|             | 10949702|
| Polar Cyclic | 65  | 10  | 10  | 5916638| 1775373,283883.9| 330,35,125 |             | 26767383|

### Table 2: Degenerate examples

Three random LP models were used. The Kuhn & Quandt polytopes [11] were constructed by generating inequalities of the form $\sum_{j=1}^{d} a_{ij}x_j \leq 10,000$ with $a_{ij}$ chosen randomly from the set {0, ..., 1000}, and then adding nonegativity constraints $x_j \geq 0, j = 1...d$. Examples named Random were constructed by generating inequalities of the form $\sum_{j=1}^{d} a_{ij}x_j \leq 1$ with $a_{ij}$ chosen randomly from the set {−1000, ..., 1000}.
Table 3: Random examples

The observed time taken by disjointlp to compute $k$ disjoint paths does not increase linearly with every iteration $k = 1...d$, as illustrated in Table 4. Computing the first few paths is comparable to the time taken to solve the LP with the simplex method, while computing the last few paths is more comparable to the time taken to enumerate all the vertices via known pivot techniques, for example using lrs [2]. Consider the Kuhn-Quandt, $d = 10, n = 60, |V| = 21044$, example on which lrs takes 21427 pivots to enumerate all the vertices.

Table 4: Growth in number of pivots

5.2 Complexity

Let $k$ be the number of disjoint paths, $|V|$ the number of vertices in $G(P)$, $|B|$ the number of lex-positive bases of the LP, $d$ the dimension, and $n$ the number of inequalities in the input. We assume the bit-vector computational model allows single-operation arithmetic. When the LP is nondegenerate, $|B| = |V|$.

**Lemma 1** The $k$ vertex-disjoint monotone paths algorithm takes $O(kd^2|B|)$ pivots, and only $O(k|V|)$ pivots when the LP is nondegenerate.

**Proof.** Pivots are performed during dfs of $D_c(P)$, to extend the search path and also to backtrack. For every iteration, the dfs may build a spanning tree across all $V$ vertices which has $|V| - 1$ edges.
Each edge along this tree is traversed at most once in each direction. Similarly for every degenerate vertex, we may have to enumerate all the lex-positive bases via reverse search. If a vertex \( v \) has \( |B^v| \) bases, then the reverse search tree at \( v \) has \( |B^v| - 1 \) edges and each edge is traversed a maximum of two times. The number of pivots required to find the lex-min basis of a degenerate vertex is \( O(d) \) and we may have to do this \( O(d|B|) \) times.

The number of operations required for a single pivot is \( \theta(nd) \), and so \( O(knd^3|B|) \) time is spent pivoting, \( O(knd|V|) \) when \( |V| = |B| \). Carefully constructed worst-case examples for simplex methods, such as \([1]\) and \([10]\), illustrate that \( \Omega(|V|) \) pivots may have to be taken to find a single path from source to sink. However \([13]\) proves that the number of vertices along a strict monotone path is strictly less than \(|V|\).

**Lemma 2** The time spent on AVL tree search is \( O(kd^2|V| \log |V|) \).

**Proof.** For each iteration, the algorithm may need to perform an AVL search for every edge of \( D_c(P) \) of which there can be \( \frac{1}{2}|V|d \). The maximum number of nodes in each AVL tree is \( O(|V|) \) and so the search time is \( O(\log |V|) \). The time taken for comparison of keys in the AVL tree is \( O(d) \) as we compare the indices of cobases.

**Lemma 3** The \( k \) vertex-disjoint monotone paths algorithm requires \( O(d|V|) \) space.

**Proof.** In the worst-case all the vertices in \( D_c(P) \) will be either marked during \( dfs \), or elements of the disjoint paths computed. The storage cost of a vertex is \( O(d) \).

**Theorem 3** A collection of \( k \) internally vertex-disjoint monotone \( s-t \) paths in a \( D_c(P) \) can be found in \( O(knd^3|B|) \) time \( (O(knd|V|) \text{ when the } P \text{ is simple}) \) and \( O(d|\bar{V}_{i-1}| + d|\bar{V}_i|) \) space per iteration \( i \) where \( \bar{V}_0 \) is the empty set, \( \bar{V}_{i-1} \) is the set of vertices along the \( i-1 \) disjoint paths found so far, and \( \bar{V}_i \) is the current set of vertices marked by the search path in iteration \( i \).

When the polytope is simple, or near simple, the theoretical time for searching the AVL trees slightly dominates the time spent pivoting in the theoretical analysis. However experimental observations show that the \( k \) vertex-disjoint monotone paths algorithm is memory efficient and little time is spent searching AVL trees. Time is spent pivoting and so the this approach to computing disjoint paths works favourably when the \( LP \) is nondegenerate or contains little degeneracy. Pivoting to compute disjoint paths is much less effective when the input is highly degenerate, as expected, since pivot methods for vertex enumeration perform similarly in the presence of degeneracy \([3]\). The complexities of our algorithm are slightly worse than that of enumerating all the vertices and edges, explicitly storing \( D_c(P) \) as a edge adjacency list, and employing a network flow algorithm on the stored digraph. However in practice \textit{disjointlp} frequently computes disjoint paths faster than state-of-the-art vertex enumeration codes can enumerate all the vertices, especially if just a few disjoint paths are required.

16
References


