Reverse Search for Enumeration

David Avis *
School of Computer Science
McGill University
3480 University, Montreal
Quebec H3A 2A7, Canada

Komei Fukuda †
Graduate School of Systems Management
University of Tsukuba
3-29-1 Otsuka, Bunkyo-ku
Tokyo 112, Japan

(December 15, 1992; Revised November 3, 1993)

Abstract

The reverse search technique has been recently introduced by the authors for efficient enumeration of vertices of polyhedra and arrangements. In this paper, we develop this idea in a general framework and show its broader applications to various problems in operations research, combinatorics, and geometry. In particular, we propose new algorithms for listing

(i) all triangulations of a set of n points in the plane,

(ii) all cells in a hyperplane arrangement in R^d,

(iii) all spanning trees of a graph,

(iv) all Euclidean (non-crossing) trees spanning a set of n points in the plane,

(v) all connected induced subgraphs of a graph, and

(vi) all topological orderings of an acyclic graph.

Finally we propose a new algorithm for the 0-1 integer programming problem which can be considered as an alternative to the branch-and-bound algorithm.

1 Introduction

The listing of all objects that satisfy a specified property is a fundamental problem in combinatorics, computational geometry, and operations research. Typical objects to be enumerated are spanning trees in a connected graph, vertices and faces of a convex polyhedron or an arrangement of hyperplanes given by a system of linear inequalities, triangulations of a set of points in the plane, etc.

There are several known search techniques for enumeration problems. Backtrack search is known to be useful for various enumeration problems associated with graphs [18]. For enumeration problems in computational geometry, the incremental search technique has been frequently used [7]. Graph search such as depth first search or breadth first search can be widely applicable for the case where the objects to be listed are the vertices of some connected graph.

In this paper, we introduce a new exhaustive search technique, called reverse search, which can be considered as a special graph search. This new search can be used to design efficient algorithms for various enumeration problems such as those mentioned above. Reverse search algorithms, if successfully designed, have the following characteristics:

*Research supported by the Natural Science and Engineering Research Council Grant number A3013 and the F.C.A.R. Grant number EQ1678.

†Partially supported by (1) Grant-in-Aids for Co-operative Research (03832008) of the Ministry of Education, Science and Culture and (2) Fujitsu Laboratories Ltd. Kawasaki, Japan.

(1) time complexity is proportional to the size of output times a polynomial in the size of input,

(2) space complexity is polynomial in the size of input,

(3) parallel implementation is straightforward, (since the procedure can be decomposed into multiple independent subprocedures at each general stage).

In order to explain the basic idea of reverse search, let G be a connected graph whose vertices are precisely the objects to be listed, and suppose we have some objective function to be maximized over all vertices of G. A local search algorithm on G is a deterministic procedure to move from any vertex to some neighboring vertex which is larger with respect to the objective function until there exists no better neighboring vertex. (Note that a local search algorithm will be defined as a more general procedure in the formal discussion in Section 2). A vertex without a better neighboring vertex is called local optimal. The algorithm is finite if for any starting vertex, it terminates in a finite number of steps. Well-known examples of local search algorithms are the simplex method for linear programming, the edge-exchange algorithm for finding a minimum spanning tree in a weighted graph [1, Section 10.5], and the flip algorithm for finding a Delaunay triangulation in the plane [7, 8, 21]. The simplex method is not finite in general, but finite if a certain pivot rule such as Bland's smallest subscript rule is used to restrict the pivot selection, while the other two algorithms are finite (A detailed description of the flip algorithm will be given in Section 3).

Let us imagine the simple case that we have a finite search algorithm and there is only one local optimal vertex x^* (which is the optimal solution). Consider the digraph T with the same vertex set as G and the edges which are all ordered pairs (x, x') of consecutive vertices x and x' generated by the local search algorithm. It should be clear that T is a tree spanning all vertices with the only sink x^*. Thus if we trace this graph T from x^* systematically, say by depth-first search, we can enumerate all vertices (i.e. objects). The major operation here is tracing each edge against its orientation which corresponds to reversing the local search algorithm, while the minor work of backtracking is simply performing the search algorithm itself. It is noteworthy that we do not have to store any information about visited vertices for this search because T is itself a tree.

This new search technique has an interesting application to hard combinatorial optimization. Observe that for each vertex x, every vertex y below x in T (those y such that there is a directed path from y to x) has no larger objective value. Suppose we are looking for some vertex satisfying a side constraint (e.g., integrality for linear programming case) with largest objective value. Then, one can perform a reverse search but only partially: Keep the current best solution $\overline{x}$ and the current best value $\overline{z}$, and whenever the search detects a better solution satisfying the side constraint, update the current best solution and value. Whenever it detects a vertex with lower objective value, then abandon going lower in the tree.

We should make some remarks on parallelization of the reverse search. It is quite easy to see that a reverse search algorithm can be easily parallelized, since it only has to visit all vertices of a well-defined tree from a given root. One trivial implementation is to assign some free processor a son of the root whose branch is not yet traversed. This can be done recursively of course: each assigned processor also assigns some of its sons to any free processors. The termination of each sub-task is easily recognized by using a depth counter. The important question is: how much can we accelerate the computation? Obviously, it is restrained by the height of the tree from the root, and the computational time depends at least linearly on this height. While we cannot easily estimate the height in some cases like the simplex method case, there are many cases where the height is small. We believe that such cases have significant potential for successful parallel computing. Among others, these cases include the enumeration of spanning trees in a connected graph, vertices and cells in an arrangement of hyperplanes, and triangulations of a point set.

The original idea of reverse search came from the vertex enumeration algorithm [4, 5], proposed by the authors, for polyhedra or for arrangements of hyperplanes which reverses the simplex algorithm with Bland's smallest subscript rule or the criss-cross method for linear programming, respectively.

Here is how the present paper is organized. The next section is devoted to a formal presentation of local search and reverse search. In Section 3, we give several applications of reverse search. The notion of partial reverse search is given in Section 4 together with some applications such as 0-1-integer programming.

2 Reverse Search

In the introduction we have given a basic idea of reverse search for enumeration. Here we shall present it formally with more generality.

Let $G = (V, E)$ be a (undirected) graph with vertex set V and edge set E. We shall call a triple (G, S, f) *local search* if S is a subset of V, f is a mapping: $V \backslash S \Rightarrow V$ satisfying

(L1) $\{v, f(v)\} \in E$ for each $v \in V \backslash S$.

and *finite* local search if in addition,

(L2) for each $v \in V \backslash S$, there exists a positive integer k such that $f^k(v) \in S$.

Here is a procedural form of a local search (G, S, f):

procedure LocalSearch(G,S,f,v_0:vertex of G);
  $v := v_0$;
  while $v \notin S$ **do**
    $v := f(v)$
  endwhile;
  output v.

The function f is said to be the *local search function*, and G the *underlying graph structure*. Naturally, we consider the set V to be the set of *candidates* for a solution, the set S to be the set of *solutions*. The local search function f is simply an algorithm for finding one solution.

It is not difficult to find examples of local search. To list a few,

- the simplex method for linear programming, where V is the set of feasible bases, E is induced by the pivot operation, f is the simplex pivot, and S is the set of optimal bases,
- the edge-exchange algorithm for finding a minimum spanning tree in a weighted graph [1, Section 10.5], where V is the set of all spanning trees, E is induced by the edge-exchange operation, f is the best-improvement exchange algorithm, and S is the set of all minimum spanning trees,
- the flip procedure for finding a Delaunay triangulation in the plane for a given set of points, where V is the set of all possible triangulations, E is induced by the flip operation, f is the flip algorithm, and S is the set of Delaunay triangulations.

It will be helpful for us to keep at least one of these examples in mind for better understanding of several new notions to be introduced below.

The *trace* of a local search (G, S, f) is a directed subgraph $T = (V, E(f))$ of G where

$$E(f) \quad = \quad \{(v, f(v)) : v \in V \backslash S\}.$$

The trace T is simply the digraph with all vertices of G and those edges of G used by the local search. We also define the *height* $h(T)$ of a trace T as the length of a longest directed path in T. An obvious but important remark is

Property 2.1. *If (G, S, f) is a finite local search then its trace T is a directed spanning forest of G with each component containing exactly one vertex of S as a unique sink.*

Let (G, S, f) be a finite local search with trace T, and denote by $T(s)$ the component of T containing a vertex s for each $s \in S$. We call the following procedure an abstract reverse search:

procedure AbstractReverseSearch(G,S,f);
  for each vertex $s \in S$ **do**
    traverse the component $T(s)$ and output all its vertices
  endfor.

Here we are purposely vague in describing how we traverse $T(s)$. The actual traversal depends on how the local search is given: in almost all cases for which reverse search is useful, G is not explicitly given. Also, we shall deal with the case where $|S| > 1$, and the set S is not explicitly given. In many cases, we can consider S to be the output of another reverse search for which the solution set is a singleton.

Thus, it is extremely useful to discuss a special implementation of reverse search when the local search is given in a certain way which is general enough for our applications to be described later but yet restricted enough for us to make interesting statements about the time complexity of reverse search.

We say that a graph G is given by *adjacency-oracle* or simply *A-oracle* when the following conditions are satisfied:

(A1) The vertices are represented by nonzero integers.

(A2) An integer δ is explicitly given which is an upper bound on the maximum degree of G, i.e., for each vertex $v \in V$, the degree $deg(v)$ is at most this number.

(A3) the adjacency list oracle *Adj* satisfying (i), (ii) and (iii) is given:

- **(i)** for each vertex v and each number k with $1 \leq k \leq \delta$ the oracle returns $Adj(v, k)$, a vertex adjacent to v or extraneous 0 (zero),
- **(ii)** if $Adj(v, k) = Adj(v, k') \neq 0$ for some $v \in V$, k and k', then $k = k'$,
- **(iii)** for each vertex v, $\{Adj(v, k) : Adj(v, k) \neq 0, 1 \leq k \leq \delta\}$ is exactly the set of vertices adjacent to v.

The conditions (i) - (iii) imply that *Adj* returns each adjacent vertex to v exactly once during the δ inquiries $Adj(v, k)$, $1 \leq k \leq \delta$, for each vertex v.

The conditions (A2) and (A3) may not seem to be natural, but as we will see, in many cases, we have no knowledge of the maximum degree of the underlying graph but only an upper bound.

Consider the simplex method. For each feasible basis, some pivot operations lead to feasible bases, and others lead to non-feasible bases. In general (with possible degeneracy and an unbounded feasible region), we do not know the maximum number of adjacent feasible bases. However, we have a trivial bound, that is, the number of pivot positions (= number of basic variables times number of nonbasic variables). Associated with each feasible basis and each k-th pivot position we have either an adjacent feasible basis or something else (i.e. a nonfeasible basis or impossible pivot), that determines our A-oracle. For the flip algorithm, one can naturally see that the underlying graph is given by A-oracle.

A local search (G, S, f) is said to be *given by an A-oracle* if the underlying graph G is.

When a local search is given by an A-oracle, we can write a particular implementation of abstract local search. The following procedure, ReverseSearch, which will be used in all of the applications in the present paper, is the one where the traversal of each component is done by depth first search and the set S is explicitly given:

```
     procedure ReverseSearch(Adj,δ,S,f);
         for each vertex s ∈ S do
             v := s; j := 0; (* j: neighbor counter *)
             repeat
                 while j < δ do
                     j := j + 1;
(r1)                 next := Adj(v, j);
                     if next ≠ 0 then
(r2)                     if f(next) = v then  (* reverse traverse *)
                             v := next; j := 0
                         endif
                     endif
                 endwhile;
                 if v ≠ s then (* forward traverse *)
(f1)                 u := v;    v := f(v);
(f2)                 j := 0;    repeat j := j + 1 until Adj(v, j) = u (* restore j *)
                 endif
             until v = s and j = δ
         endfor.
```

Note that for each vertex $v \in V \backslash S$, exactly one "forward traverse" is performed in the procedure ReverseSearch. The time complexity of ReverseSearch can be now evaluated. For a local search (G, S, f) given by an A-oracle, let $t(f)$ and $t(Adj)$ denote the time to evaluate f and Adj, respectively.

Theorem 2.2. *Suppose that a local search (G, S, f) is given by an A-oracle. Then the time complexity of ReverseSearch is $O(\delta\, t(Adj)|V| + t(f)|E|)$.*

Proof. It is easy to see that the time complexity is determined by the total time spent to execute the four lines (r1), (r2), (f1) and (f2). The first line (r1) is executed at most δ times for each vertex, and the total time spent for (r1) is $O(\delta\, t(Adj)|V|)$. The line (r2) is executed as many times as the degree $deg(v)$ for each vertex v, and thus the total time for (r2) is $O(t(f)|E|)$. The third line (f1) is executed for each vertex v in $V \backslash S$, and hence the total time for (f1) is $O(t(f)(|V| - |S|))$. Similarly, the total time for (f2) is $O(\delta\, t(Adj)(|V| - |S|))$. Since $|V| - |S| \leq |E|$, by adding up the four time complexities above, we have the claimed result. ∎

Corollary 2.3. *Suppose that a local search (G, S, f) is given by an A-oracle. Then the time complexity of ReverseSearch is $O(\delta\ (t(Adj) + t(f))|V|)$. In particular, if δ, $t(f)$ and $t(Adj)$ are independent of the number $|V|$ of vertices in G, then the time complexity is linear in the output size $|V|$.*

Proof. The claim follows immediately from Theorem 2.2 and the fact that $2|E| \le \delta|V|$. ∎

The assumption that δ, $t(f)$ and $t(Adj)$ are independent of the number $|V|$ is not satisfied in general (e.g. when G is a complete graph), but is for many cases it can be assumed. In fact, all applications to be presented in the next section satisfy this assumption.

We have already two simple formulas, Theorem 2.2 and Corollary 2.3, for the time complexity of the reverse search. We shall use these to evaluate the time complexity of some of our applications. However, a stronger result is possible in some cases. One of such cases is when the lines (r1) and (r2) have a shortcut, i.e., it is possible to check for any vertex v of G and any integer $1 \le j \le \delta$ whether $f(Adj(v, j)) = v$ without executing Adj and f. Another case is when the procedure (f2) has a shortcut, i.e., it is possible for any vertex v of G to determine the integer j such that $Adj(f(v), j) = v$ without executing Adj explicitly. In order to deal with these cases more clearly we shall give below an alternative version of reverse search. Here we use the convention that $f(0) = 0$.

```
        procedure ReverseSearch2(Adj,δ,S,f);
            for each vertex s ∈ S do
                v := s; j := 0; (* j: neighbor counter *)
                repeat
                    while j < δ do
                        j := j + 1;
(r1')                   if f(Adj(v, j)) = v then  (* reverse traverse *)
(r2')                       v := Adj(v, j); j := 0
                        endif
                    endwhile;
                    if v ≠ s then  (* forward traverse *)
(f1)                    u := v;    v := f(v);
(f2')                   determine j such that Adj(v, j) = u (* restore j *)
                    endif
                until v = s and j = δ
            endfor.
```

In order to describe the time complexity of this procedure, we define $t^R(Adj, f)$ to be the time necessary to decide for any vertex v of G and any integer $1 \le j \le \delta$ whether $f(Adj(v, j)) = v$ (i.e., $t^R(Adj, f)$ is the time to decide whether moving from v to $Adj(v, j)$ is a reverse of f). Similarly, we define $t^F(Adj, f)$ to be the time necessary for any vertex v of G to determine the integer j such that $Adj(f(v), j) = v$.

Theorem 2.4. *Suppose that a local search (G, S, f) is given by an A-oracle. Then the time complexity of ReverseSearch2 is $O((t(Adj) + \delta\ t^R(Adj, f) + t(f) + t^F(Adj, f))|V|)$.*

Proof. The proof is similar to that of Theorem 2.2. The total time for (r1') is $O(\delta\ t^R(Adj, f)|V|)$, and that for (r2') is $O(t(Adj)(|V| - |S|))$. Similarly, the total time for (f1) is $O(t(f)(|V| - |S|))$, and that for (f2') is $O(t^F(Adj, f)(|V| - |S|))$. Adding up all these yields the result. ∎

Looking into this theorem, we notice a possibility of further refinement of reverse search. Remark that the part $t_1 = (t(Adj) + \delta\, t^R(Adj, f))$ of the time complexity is the time necessary for the reverse traversal, i.e., moving away from the top vertex, and the remaining part $t_2 = (t(f) + t^F(Adj, f))$ is the time for the forward traversal, i.e. , moving toward the top vertex. One cannot really shorten the first part, but interestingly one can shorten the latter part by storing the forward traverse paths. More precisely, if we store the forward sequence to return to the top vertex while reversing, Neither f nor Adj need to be evaluated to go forward. This remark can be particularly important when the trace of a local search has a short height and t_1 is an order of magnitude smaller than t_2, though presently we do not have any such applications.

Finally we should remark that the space complexity is usually independent of the cardinality of the output. At the moment, we cannot evaluate precisely the space complexity, but it only depends linearly on the space necessary to store a single vertex and the space necessary to realize the functions (oracles) f and Adj.

3 Applications of Reverse Search

3.1 Vertex Enumeration in Polyhedra

The vertex enumeration problem is to list all vertices of the convex polyhedron given by $P = \{x : Ax \leq b, x \geq 0\}$, where A is an $m \times n$ matrix and b is a m-vector.

Consider the linear program of form: maximize cx subject to $Ax \leq b$ and $x \geq 0$. The simplex algorithm can be considered as a finite local search (G_{LP}, S_{LP}, f_{LP}) where $G_{LP} = (V_{LP}, E_{LP})$ is a graph with V_{LP} is the set of all feasible bases; where two bases are adjacent if and only if one can be obtained from the other by a pivot operation; S_{LP} being the set of all optimal bases; and f_{LP} is the simplex algorithm with Bland's smallest subscript rule. Moreover, represent each basis by the set of indices of basic variables, and define Adj_{LP} to be $Adj_{LP}(B, (i, j)) = B - i + j$ if B is a basis and $B - i + j$ is a basis, and 0 otherwise, for each basic and nonbasic indices i and j. Then the local search is given by an A-oracle Adj_{LP} with $\delta_{LP} = m \times n$.

Now, how can we find all vertices of P? We can easily find one feasible basis of the linear inequality system $Ax \leq b$ and $x \geq 0$, say B, by the simplex method or the interior-point method. Then we can set up an LP with objective function cx for which the current basic solution is the unique optimal solution and B is an optimal basis. The associated local search (G_{LP}, S_{LP}, f_{LP}) immediately yields the reverse search ReverseSearch(Adj_{LP},δ_{LP},S_{LP},f_{LP}) to list all feasible bases as long as the set S_{LP} of all optimal bases are explicitly given.

If the set S_{LP} is the singleton $\{B\}$ and we are done. Otherwise, we can enumerate all optimal bases from B by another reverse search with respect to the dual simplex method applied to an auxiliary problem. See [4] for details.

If the system $Ax \leq b, x \geq 0$ is nondegenerate, then one can design a much simpler algorithm. The critical difference is that for each feasible basis B and each nonbasic index j, there exists at most one basic index $i = i(j)$ such that $B - i + j$ is again a feasible basis. Define Adj_{LP} to be $Adj_{LP}(B, j) = B - i + j$ if there exists i such that $B - i + j$ is a feasible basis, and 0 otherwise, for each nonbasic index j. Then the local search is given by an A-oracle Adj_{LP} with smaller $\delta_{LP} = n$.

It is well known that the simplex method with Bland's rule might take an exponential number of pivots to find an optimal solution, see [3]. This means that the height of the trace T_{LP} cannot be bounded by a polynomial function of m and n. Theoretically, this means even the best parallel implementation may not be much faster. The expected behavior, however, might turn out to be quite different.

A *Mathematica* implementation of the vertex enumeration algorithm is available in [12], and a C implementation in [2].

3.2 Enumeration of Cells in Arrangements

Let $E = \{1, 2, \ldots, m\}$, let $\mathcal{A}$ be an arrangement of distinct hyperplanes $\{H_i : i \in E\}$ in R^n, where each hyperplane is given by a linear equality $H_i = \{x : a^i x = b_i\}$. The two sides of H_i are $H_i^+ = \{x : a^i x \geq b_i\}$ and $H_j^- = \{x : a^j x \leq b_j\}$. For each $x \in R^n$, the signed vector $SV(x)$ of x is the vector in $\{-, 0, +\}^E$ defined by

$$SV(x)_i = \begin{cases} - & \text{if } x \in H_i^- \\ 0 & \text{if } x \in H_i \\ + & \text{if } x \in H_i^+ \end{cases} \qquad (i \in E).$$

Let V_{CELL} be the set of signed vectors of points in R^n whose nonzero support is E. We can identify each vector c in V_{CELL} with the open cell (open n-face) of the arrangement defined by $\{x : SV(x) = c\}$. For two cells c and c', let $sep(c, c')$ be the set of separators of c and c', that is, the set of elements i of E such that c_i and c'_i have opposite signs. We say that two cells c and c' are *adjacent* in G_{CELL} if they differ in only one component, or equivalently, $|sep(c, c')| = 1$. The following lemma is important.

Lemma 3.1. *For any two distinct cells c and c' in V_{CELL}, there exists a cell c'' which is adjacent to c and $sep(c, c'') \subset sep(c, c')$.*

Proof. Let c and c' be two distinct cells, and let x (x') be a point in c (in c', respectively) in general position. Moving from x toward x' on the line segment $[x, x']$, we encounter the sequence of cells: $c_o = c, c_1, c_2, \ldots, c_k = c'$, and we can easily verify that c_1 is adjacent to c and $sep(c, c_1) \subset sep(c, c')$. ∎

Let us assume that V contains the cell c^* of all $+$'s. Lemma 3.1 implies that for each cell c different from c^*, there is a cell c'' which is adjacent to c and $sep(c^*, c'') \subset sep(c^*, c)$. Let us define $f_{CELL}(c)$ as such c'' that is lexico-largest (i.e., the unique element in $sep(c, c'')$ is smallest possible). Then, $(G_{CELL}, S_{CELL}, f_{CELL})$ is a finite local search with $S_{CELL} = \{c^*\}$. By Lemma 3.1, one immediately obtains:

Corollary 3.2. *The height of the trace T_{CELL} of the local search $(G_{CELL}, S_{CELL}, f_{CELL})$ is at most m.*

Figure 4.1 describes the trace of the local search on a small example with $n = 2$ and $m = 4$.

By reversing this local search, we obtain an algorithm to list all cells in an arrangement. There are a few things to be explained for an implementation. First, we assumed that the cell c^* of all $+$'s is given, but we can pick up any cell c in the arrangement, and consider it as the cell of all $+$'s since replacing some equality $a^i x = b_i$ by $-a^i x = -b_i$ does not essentially change the arrangement. Note that one can obtain an initial cell by picking up any random point in R^n and perturbing it if it lies on some hyperplanes.

Now, how can we realize ReverseSearch(Adj_{CELL},δ_{CELL},S_{CELL},f_{CELL}) in an efficient way? First we can set $\delta_{CELL} = m$ and $S_{CELL} = \{c^*\}$. For any cell $c \in V_{CELL}$ and $k \in E$, the function $Adj_{CELL}(c, k)$ can be realized via solving an LP of the form

$$\begin{array}{ll} \text{minimize (maximize)} & y_k \\ \text{subject to} & y = Ax - b, \\ & y_i \geq 0 \text{ for all } i \neq k \text{ with } c_i = +, \\ & y_i \leq 0 \text{ for all } i \neq k \text{ with } c_i = -, \end{array} \tag{3.1}$$

where minimization (maximization) is chosen when $c_k = +$ ($c_k = -$, respectively). The function returns the adjacent cell c' with $sep(c, c') = \{k\}$ if and only if LP (3.1) has a feasible solution with negative (positive) objective value. The time $t(Adj_{CELL})$ depends on how an LP with n variables and $m-1$ inequalities is solved. We denote this as a function $l(m,n)$ of m and n.

There is a straightforward implementation of f_{CELL}, which solves a sequence of LP's similar to (3.1) with objective functions $y_1, y_2, y_3, \ldots$. This means we may have to solve $O(m)$ LP's in the worst case. Presently we don't know how to implement it in a more efficient manner.

Theorem 3.3. *There is an implementation of ReverseSearch(Adj_{CELL}, δ_{CELL}, S_{CELL}, f_{CELL}) for the cell enumeration problem with time complexity $O(m\ n\ l(m,n)|V_{CELL}|)$ and space complexity $O(m\ n)$.*

Proof. To prove this, first we recall that Theorem 2.2 says, the time complexity of ReverseSearch is $O(\delta\ t(Adj)|V| + t(f)|E|)$. As we remarked earlier, $\delta_{CELL} = m$, $t(Adj_{CELL}) = O(l(m,n))$, and $t(f_{CELL}) = O(m\ l(m,n))$. Since $|E_{CELL}| \leq n\ |V_{CELL}|$ holds for any arrangement (see, e.g., [10, 11]), the claimed time complexity follows. The space complexity is clearly same as the input size $O(m\ n)$. ∎

We believe that there is no previously known algorithm to enumerate all cells of an arrangement whose time complexity is polynomial in the size of output.

The cardinality of output is of course exponential in m and n, and the maximum, explicitly given by Buck's formula $\sum_{i=0}^{n} \binom{m}{i}$, is attained for any simple arrangements, see [6, 7]. By Corollary 3.2, the cell enumeration can profit a lot from parallel implementation.

3.3 Enumeration of Triangulations

Let P be a set $\{p_1, \ldots, p_n\}$ of n distinct points in the plane. A pair $\{p, q\}$ of distinct points in P is called an *edge* if the line segment connecting p and q does not contain any other point of P. A triple $\{p, q, r\}$ of points in P is called a *triangle* if they are not collinear and the their convex hull (triangle region) does not contain any other points in P. A point or edge in P is called *external* if it is contained in the boundary of the convex hull of P, and *internal* otherwise.

A *triangulation* of P is a set Δ of triangles in P such that (1) each external edge is contained in exactly one triangle of Δ, (2) each internal edge is contained in either no triangle of Δ or exactly two triangles of Δ. An *edge* of a triangulation Δ is an edge contained in at least one triangle of Δ.

By using Euler's relation, one can easily see that the number of triangles and edges of a triangulation are independent of the choice of triangulation.

Proposition 3.4. *Let Δ be a triangulation of P, and let f_1 and f_2 be the number of edges and triangles of Δ, respectively. Then, they are determined by $f_1 = 3n - n^* - 3$, $f_2 = 2n - n^* - 2$, where n^* denotes the number of external points.*

It is clear from the definition that the number of triangulations is finite. The enumeration of all possible triangulations of P is the problem in this section. In order to apply the reverse search technique, the notion of Delaunay triangulation and the flip algorithm is very useful.

Let Δ be a triangulation with f_2 triangles whose interior angles $\alpha_1, \alpha_2, \ldots, \alpha_{3f_2}$ are indexed in such a way that $\alpha_i \leq \alpha_j$ for any $1 \leq i < j \leq 3f_2$. The vector $\alpha(\Delta) = (\alpha_1, \alpha_2, \ldots, \alpha_{3f_2})$ is called the angle vector of Δ . A triangulation is said to be *Delaunay* if its angle vector is lexicographically maximal over all possible triangulations of the same point set, where the comparison of components is done from left to right.

Let Δ be a triangulation. Let $\{a,b\}$ be any internal edge of Δ, and let $\{a,b,c\}$ and $\{a,b,d\}$ be the two triangles of Δ containing it. We call $\{a,b\}$ *flippable* if the set $Flip(\Delta,\{a,b\}) := \Delta \setminus \{\{a,b,c\},\{a,b,d\}\} \cup \{\{a,c,d\},\{b,c,d\}\}$ is again a triangulation of P. One can easily see that an internal edge $\{a,b\}$ is flippable if and only if the points a,b,c,d form a convex quadrangle.

We call $\{a,b\}$ *legal* if the circumscribing disk of one of the triangles abc, abd does not contain the other, and *illegal* otherwise. When an edge $\{a,b\}$ is illegal, it is always flippable and the operation $Flip(\Delta,\{a,b\})$ is called a *Delaunay flip*.

It is known that a triangulation Δ is Delaunay if and only if it does not contain any illegal edges. The flip algorithm is simply a procedure to use the Delaunay flip operation repeatedly in any order until no such operation is possible, see Figure 4.2. The following theorem states that the flip algorithm is finite.

Theorem 3.5 ([8, 21]) : *The flip algorithm terminates in $O(n^2)$ steps and finds a Delaunay triangulation of P, starting with any initial triangulation.*

To apply reverse search, let V_{TRI} (S_{TRI}) be the set of all (Delaunay, respectively) triangulations of P. The underlying graph G_{TRI} is (V_{TRI}, E_{TRI}) where two vertices are adjacent if and only if one is a flip of the other. We define a local search f_{TRI} as the function from $V_{TRI} \setminus S_{TRI}$ to V_{TRI} such that

$$f_{TRI}(\Delta) := Flip(\Delta,\{a,b\})$$

where $\{a,b\}$ is the lexico-smallest illegal edge of Δ.

Now, how one should design an A-oracle? For any triangulation Δ, let L be the list of interior edges ordered lexicographically and let L_k be the kth member of L. By Proposition 3.4, the cardinality $|L|$ of L is exactly $3n - 2n^* - 3$, which we denote by δ_{TRI}. The adjacency oracle $Adj_{TRI}(\Delta, k)$ is then defined as

$$Adj_{TRI}(\Delta,k) := \begin{cases} Flip(\Delta, L_k) & \text{if } L_k \text{ is flippable,} \\ 0 & \text{otherwise,} \end{cases}$$

for each $k = 1, \ldots, \delta_{TRI}$.

Theorem 3.6. *There is an implementation of ReverseSearch2(Adj_{TRI}, δ_{TRI}, S_{TRI}, f_{TRI}) for the triangulation enumeration problem with time complexity $O(n\,|V_{TRI}|)$ and space complexity $O(n)$.*

Proof. For the implementation, we can use the quad-edge data structure [13] for storing a triangulation. Also we store L as a linked list of edges each with a flag indicating either nonflippable, legal or illegal, and store the lexico-smallest illegal edge of L. For the analysis of time complexity, we apply Theorem 2.4. First, note that we can evaluate Adj_{TRI} and f_{TRI} in $O(n)$ time, including time to update L and the triangulation data. Since we store the lexico-smallest illegal edge of L, $t^R(Adj, f)$ and $t^F(Adj, f)$ are both $O(1)$. Since $\delta_{TRI} = O(n)$ and by Theorem 2.4, we have the stated time complexity. The space complexity is clearly $O(n)$. ∎

We remark that enumerating all elements in S_{TRI} (i.e. the enumeration of all Delaunay triangulations) is unnecessary. It is possible to transform any Delaunay triangulation to another by a sequence of flips, and one can extend the local search f_{TRI} so that it finds the lexico-smallest Delaunay triangulation by flipping some legal edge at a non-lexico-smallest Delaunay triangulation.

Theorem 3.5 shows that a parallel implementation can be quite fast.

It should be mentioned that essentially the same algorithm has been discovered independently by H. Telley.

3.4 Enumeration of Connected Induced Subgraphs

Let $G = (V, E)$ be a graph with vertex set $V = \{1, 2, \ldots, n\}$ and edge set E of size m. For any subset U of V, we denote by $G(U) = (U, E(U))$ the subgraph of G induced by U, i.e. $E(U)$ is the set of edges of G whose endpoints are both in U.

In this section, we apply the reverse search technique to the enumeration of all connected induced subgraphs of a given graph G. It will be seen that the enumeration of connected subgraphs as opposed to connected "induced" subgraphs can be treated in a similar manner.

The following lemma is essential.

Lemma 3.7. *Let $G = (V, E)$ be a graph and let U be a nonempty subset of V such that $G(U)$ is connected. Then there exists a vertex $j \in U$ such that $G(U - j)$ is connected.*

Proof. Let the assumptions be satisfied. If $|U| = 1$ then the lemma is trivial. Assume that $|U| \geq 2$. Take any spanning tree T of $G(U)$, and take any vertex j of T having degree one, which always exists. Removal of such a vertex cannot disconnect T and the graph $G(U)$. ∎

It is quite easy to prove that there is an opposite operation preserving connectivity.

Lemma 3.8. *Let $G = (V, E)$ be a connected graph and let U be a proper subset of V such that $G(U)$ is connected. Then there exists a vertex $j \in V \setminus U$ such that $G(U + j)$ is connected.*

Each of the two lemmas above yields a reverse search algorithm for enumerating all connected induced subgraphs. Here we exploit the first one, Lemma 3.7.

Let V_{CIS} be the family of subsets U of V such that $G(U)$ is connected, and let $S_{CIS} = \{\emptyset\}$. By Lemma 3.7 The following local search function f_{CIS} from $V_{CIS} \setminus S_{CIS}$ to V_{CIS} is well-defined:

$$f_{CIS}(U) := U - j$$

where j is the smallest vertex in U such that $G(U - j)$ is connected. Thus for any nonempty set $U \in V_{CIS}$ the function f_{CIS} generates a unique sequence of subsets $U, f(U), f^2(U), \ldots, f^{|U|}(U) \equiv \emptyset$. The reverse search algorithm we describe here merely reverses this finite algorithm.

Now the underlying graph $G_{CIS} = (V_{CIS}, E_{CIS})$ is rather straightforward; two subsets U and U' are adjacent in G_{CIS} if and only if one is a proper subset of the other and they differ in exactly one element. The adjacency oracle A_{CIS} is then defined by

$$Adj_{CIS}(U, k) := \begin{cases} U - k & \text{if } k \in U \text{ and } U - k \in V_{CIS} \\ U + k & \text{if } k \in V \backslash U \text{ and } U + k \in V_{CIS} \\ 0 & \text{otherwise,} \end{cases}$$

for each $U \in V_{CIS}$ and each vertex $k = 1, 2, \ldots, n$. Thus we have $\delta_{CIS} = n$.

A simple implementation of the adjacency oracle Adj_{CIS} and f_{CIS} gives the following complexity of the reverse search algorithm, which one might be able to improve by using a more sophisticated data structure.

Theorem 3.9. *There is an implementation of ReverseSearch(Adj_{CIS}, δ_{CIS}, S_{CIS}, f_{CIS}) for the connected induced subgraph enumeration problem with time complexity $O(m\ n\ |V_{CIS}|)$ and space complexity $O(m + n)$.*

Proof. The essential part of implementing Adj_{CIS} and f_{CIS} is to list directly or indirectly all articulation points (i.e. cut vertices) in a graph. Using the depth-first-search tree (see [1, Section 7.4]), we have an implementation of Adj_{CIS} and f_{CIS} for which $t(Adj_{CIS}) = t(f_{CIS}) = O(m)$. From the time complexity of ReverseSearch in Theorem 2.3 and the fact that $\delta_{CIS} = n$, we immediately obtain the claimed time complexity. The space complexity is clearly same as the input size $O(m+n)$. ∎

Clearly the height of the reverse search tree T_{CIS} is at most n. This means that a parallel implementation the algorithm can be much faster.

It is interesting to note that this reverse search algorithm can be used with slight modification to enumerate all connected induced subgraphs with at most k vertices, for a fixed $k \leq n$. The only change will be an additional stopping rule (stop as soon as $|U| = k$) to search lower in the trace of f_{CIS}.

As we mentioned it above, Lemma 3.8 yields a reverse search algorithm as well. In this algorithm, the initial graph is G itself instead of the empty graph (assuming without loss of generality that G is connected). It is easy to see that an additional stopping rule in this search, gives an algorithms for enumerating all connected induced subgraphs with at least k vertices, for a fixed $k \leq n$.

Finally, we should note that the enumeration of all connected subgraphs of a graph as opposed to "induced" subgraphs can be done by the same approach. That is, the key lemmas, Lemma 3.7 and Lemma 3.8, have an immediate analogue in terms of connected subgraphs where edge insertion/deletion replaces vertex insertion/deletion.

3.5 Enumeration of Topological Orderings

Let $G = (V, E)$ be an acyclic digraph with vertex set $V = \{1, 2, \ldots, n\}$ and edge set E of size m. We denote by (i, j) an edge directed from i to j. A permutation $\pi = \pi_1\pi_2 \ldots \pi_n$ of V is said to be a *topological ordering* if $(i, j) \in E$ implies i appears to the left of j in π. Topological orderings are also called *linear extensions*.

It is well-known that a topological ordering of an acyclic graph G can be found in $O(m)$ time, see [1, Section 6.6]. In this section, we present a reverse search algorithm to enumerate all topological orderings efficiently.

Without loss of generality we assume that the trivial permutation (identity) π^0 is a topological ordering. Let V_{TOR} be the set of topological orderings, and let $S_{TOR} = \{\pi^0\}$.

For a permutation π of V and for any $1 \leq p < n$, the *local change* of π at i is the replacement of π with $LC(\pi, i)$, the permutation obtained from π by interchanging π_i and π_{i+1}. The local change is said to be *admissible* if $\pi_i > \pi_{i+1}$. Now we have a simple lemma.

Lemma 3.10. *Let π be a nontrivial permutation in V_{TOR}. Then π admits an admissible local change, and furthermore any admissible local change of π is in V_{TOR}.*

Proof. Let π be a nontrivial permutation in V_{TOR}. Since it is nontrivial, there exists an index $1 \leq i < n$ such that $\pi_i > \pi_{i+1}$. Take any such an index i. Since π^0 is a topological ordering, $(\pi_i, \pi_{i+1}) \notin E$ and thus $LC(\pi, i) \in V_{TOR}$. ∎

This lemma ensures that any nontrivial permutation in V_{TOR} can be replaced by a permutation in V_{TOR} which is better (closer to π^0). More specifically, the lemma enables us to define a finite local search function f_{TOR} from $V_{TOR} - \pi^0$ to V_{TOR} as:

$$f_{TOR}(\pi) := LC(\pi, s)$$

where s is the smallest index such that the local change of π at s is admissible (i.e. s is the smallest index such that $\pi_s > \pi_{s+1}$).

The underlying graph structure $G_{TOR} = (V_{TOR}, E_{TOR})$ is naturally derived from the function f_{TOR}; two permutations π and π' are *adjacent* if and only if one is obtained from the other by a local change. So we define the adjacency oracle Adj_{TOR} of the graph by

$$Adj_{TOR}(\pi, k) := \begin{cases} LC(\pi, k) & \text{if } (\pi_k, \pi_{k+1}) \notin E \text{ and } (\pi_{k+1}, \pi_k) \notin E \\ 0 & \text{otherwise,} \end{cases}$$

for each $\pi \in V_{TOR}$ and each index $k = 1, 2, \ldots, n-1$. Thus we have $\delta_{TOR} = n-1$.

The following lemma is important for an efficient implementation of the reverse search algorithm for the current problem.

Lemma 3.11. *Let π be a nontrivial permutation in V_{TOR} and let s be the smallest index such that the local change of π at s is admissible. For an index $1 \leq i < n$, the local change of π at i is a reverse of f_{TOR}, i.e., $f_{TOR}(LC(\pi, i)) = \pi$ if and only if $(\pi_i, \pi_{i+1}) \notin E$, $\pi_i < \pi_{i+1}$ and either one of the following conditions holds:*

(i) $i \leq s-1$;

(ii) $i = s+1$ and $\pi_s < \pi_{s+2}$ (implying, $s \leq n-2$).

Proof. The sufficiency is easy. We prove the necessity. Let i be an index $1 \leq i < n$ such that the local change of π at i is a reverse of f_{TOR}. Firstly, one can easily see that the conditions $(\pi_i, \pi_{i+1}) \notin E$ and $\pi_i < \pi_{i+1}$ must hold. Suppose that neither (i) nor (ii) holds. Then we have three cases (1) $i = s$, (2) $i = s+1$ and $\pi_s > \pi_{s+2}$, (3) $i > s+1$. Let $\pi' = LC(\pi, i)$.

Clearly the case (1) does not happen since the position s cannot be an admissible local change position for both π and π'. Thus either (2) or (3) must hold. Then the position s is the smallest index j such that the local change of π' at j is admissible. This contradicts the assumption that $f_{TOR}(\pi') = \pi$.

Therefore either (i) or (ii) must hold. This completes the proof. ∎

Theorem 3.12. *There is an implementation of ReverseSearch2(Adj_{TOR}, δ_{TOR}, S_{TOR}, f_{TOR}) for the topological ordering enumeration problem with time complexity $O(n\,|V_{TOR}|)$ and space complexity $O(m\,n)$.*

Proof. For an efficient implementation, we store the current permutation $\pi = \pi_1\pi_2\ldots\pi_n$ and the smallest index s such that the local change of π at s is admissible. Also we store the graph G with its incidence matrix so that the query $(i,j) \in E$? can be answered in $O(1)$ time. Observe that one can evaluate Adj_{TOR} and f_{TOR} in $O(n)$ time, including time to update π and the index s. Since we store the index s, Lemma 3.11 implies that we have $t^R(Adj, f) = O(1)$. By using the trivial time complexity $t^F(Adj, f) = O(n)$ and $\delta_{TOR} = O(n)$, Theorem 2.4 yields the claimed time complexity of ReverseSearch2. The space complexity is dominated by the storage for the incidence matrix which is $O(m\,n)$. ∎

There is an efficient backtrack algorithm to enumerate all topological orderings, see [14]. Recently, a Gray code algorithm has been proposed [17] which generates all topological orderings (and outputs only the local changes) in optimal $O(m + |V_{TOR}|)$ time and $O(n^2)$ space. Analysing the amortized complexity of the reverse search algorithm is an interesting problem, which might lead to a reverse search algorithm with optimal complexities.

3.6 Enumeration of Bases and Spanning Trees

Let P be a finite set with n elements, and let M be the set of bases of a matroid on the ground set P with rank m [22], i.e., M satisfies the *basis axioms*:

(1) each member of M is a subset of P with cardinality m, called a *basis* of M;

(2) for any two bases B and B' of M and for any $s \in B' \backslash B$, there exists an element $r \in B \backslash B'$ such that $B - r + s$ is again a basis of M.

There are simple well known examples of matroids.

Let A be a real matrix of rank m with n-column vectors $A_1, \ldots, A_n$, and let $P = \{A_1, \ldots, A_n\}$. A *basis* of A is defined as a maximal independent subset of P. Then the set $M_{Lin}(A)$ of all bases of A is a matroid. A matroid arising this way is called *linear* or *representable over the reals.*

For a graph G with the edge set P, the set $M_{ST}(G)$ of all spanning forests, each considered as the collection of its edges, is also a matroid. This matroid is known as the *cycle* matroid of G. By assigning arbitrary orientations to the edges of G, we have $M_{ST}(G) \simeq M_{Lin}(A_G)$ for the $(-1, 0, +1)$-incidence matrix A_G of G, the cycle matroid of a graph is always linear.

The problem of enumeration of bases of M does not make any sense if M is given explicitly. However, as we see from the two special cases above, we often have the following situation:

(a) the set P is explicitly given but not M;

(b) there is an efficient way to find a basis of M;

(c) there is an efficient way to decide whether a given subset B of P is a basis or not.

Under these conditions, the efficient enumeration of bases is a nontrivial problem. Like the cases we have already discussed in earlier sections, the reverse search technique can be naturally applied to this problem.

By (a), we may suppose that a basis B^* of M is given. Without loss of generality, we set $B^* = \{1, 2, \ldots, m\}$ and $P = \{1, 2, \ldots, n\}$. Let $V_{BAS} = M$ and $S_{BAS} = \{B^*\}$, and consider G_{BAS} to be the graph with vertex set V_{BAS} such that two vertices B, B' are *adjacent* if and only if they have exactly $m-1$ common elements. Now, the basis axioms (1) and (2) almost immediately yield a local search we need for the enumeration of bases. Namely we define f to be the function from $V_{BAS} \backslash S_{BAS}$ to V_{BAS} such that

$$f_{BAS}(B) := B - r + s,$$

where $s = \min\{j : j \in B^* \backslash B\}$ and $r = \max\{i : i \in B \backslash B^* \text{ and } B - i + s \text{ is a basis }\}$. From the basis axioms, this function is well defined, and always returns a basis closer to B^* in terms of Hamming distance.

In order to design an algorithm that can be readily applied to the linear matroid case, we employ the notion of tableau. For a basis B, the $B \times (P \backslash B)$-matrix $T(B) = [t_{ij} : i \in B \text{ and } j \in P \backslash B]$ defined by

$$t_{ij} = \begin{cases} 1 & \text{if } B - i + j \text{ is a basis} \\ 0 & \text{otherwise,} \end{cases} \qquad (i \in B, j \in P \backslash B).$$

is called the *tableau* of B.

The notion of tableau is commonly used for linear matroids with an explicitly given representation matrix A in which each tableau corresponds to an elementary (pivot) transformation of A. The tableau here is simply a combinatorial abstraction which only distinguishes the zero and the nonzero entries by 0 and 1.

For a given tableau $T(B)$, and a nonzero entry t_{rs}, the operation of replacing $T(B)$ by $T(B-r+s)$ is called a *pivot on* (r, s), denoted by $Piv(B, (r, s))$. Since this operation is basic in linear cases, it is useful to implement our algorithm using the pivot operation as an elementary operation. Let us consider $t(Piv)$ to be the time necessary to do one pivot operation. In the linear case $M = M(A)$, we have $t(Piv) = O(mn)$.

Our adjacency oracle Adj_{BAS} is merely a disguise of the tableau:

$$Adj_{BAS}(B, (i, j)) := \begin{cases} B - i + j & \text{if } t_{ij} = 1 \\ 0 & \text{if } t_{ij} = 0 \end{cases} \qquad (i \in B, j \in P \backslash B).$$

Here, we consider $\delta_{BAS} = m \times (n-m)$, which is the number of candidates for (i, j).

For the implementation of reverse search we propose here, we maintain the tableau for the current basis B. In addition, we maintain three additional items associated with B so that we can evaluate f_{BAS}, Adj_{BAS} in constant time $O(1)$. The first one is simply the pivot position (r, s) chosen by f_{BAS} at B. The second item is the largest integer l such that $\{1, \ldots, l\} \subseteq B$, which will be denoted by $last(B)$. The last one is the reversibility flag vector $R(B) = [R(B)_j : j \in P \backslash B]$ given by

$$R(B)_j := \begin{cases} \textbf{true} & \text{if } j > m \text{ and } t_{kj} = 0 \text{ for all } k \in B \text{ with } k > j \\ \textbf{false} & \text{otherwise,} \end{cases} \qquad (j \in P \backslash B).$$

Then we have the following lemma.

Lemma 3.13. *For a basis B, a position (i, j) in the tableau $T(B)$ is a reverse pivot position with respect to f_{BAS} if and only if $t_{ij} \neq 0$, $i \leq last(B)$ and $R(B)_j =$***true***.*

Proof. Left to the reader. ∎

Theorem 3.14. *There is an implementation of ReverseSearch2(Adj_{BAS}, δ_{BAS}, S_{BAS}, f_{BAS}) for the basis enumeration problem with time complexity $O((mn + t(Piv))|V_{BAS}|)$ and space complexity independent of $|V_{BAS}|$.*

Proof. To prove this, we use Theorem 2.4 which claims the time complexity of ReverseSearch2 is $O((t(Adj) + t(f) + \delta\, t^R(Adj, f) + t^F(Adj, f))|V|)$.

For the claimed implementation, we use the data structure (the tableau and the three associated items) described above. Thus we have $O(t(f_{BAS})) = O(t(Adj_{BAS})) = O(1)$ excluding the data structure update. Clearly, $t^F(Adj, f) = O(1)$. Also, by the previous lemma, we have $t^R(Adj, f) = O(1)$. Since $\delta_{BAS} = m \times (n-m)$, the total time for enumeration excluding the time of data structure update is $O(mn|V_{BAS}|)$

Now, the time of updating a tableau and the associated data is $t(Piv) + O(mn)$. Since we must update the tableau and the associated data each time we move to a different basis, the total time for updating data is $O((t(Piv) + O(mn))|V_{BAS}|)$. This proves the theorem. ∎

For the case of spanning trees of graphs, the enumeration problem can be solved using backtrack search with time complexity $O((m+n)|M|)$, see [18]. It has been shown recently in [15, 20] that by using sophisticated data structures one can design reverse search algorithms with the optimal complexity $O(m+n+n|M|)$. While [15] describes an implementation with optimal space complexity $O(m+n)$, the algorithm in [20] can be used to scan all spanning trees in $O(m+n+|M|)$ time and $O(mn)$ space.

One can easily see that the maximum cardinality of output is $\binom{n}{m}$. In contrast, the trace of the reverse search has an exceptionally short height of at most m. This example is perhaps an ideal example of reverse search that can profit substantially from parallel implementation.

Note that this reverse search can be applied to the enumeration of vertices in arrangements of hyperplanes. This is superior to the reverse search method given in [4] in the sense of both time complexity and parallel acceleration.

3.7 Enumeration of Euclidean Spanning Trees

Let $P = \{p_1 \ldots p_n\}$ be a set of n points in the plane, no three of which are collinear. We consider trees with vertices in P and edges given by line segments with endpoints in P. Two such edges with all endpoints distinct are said to *cross* if the corresponding line segments intersect. By the general position assumption this intersection point must be at an interior point of both segments.

An *Euclidean Spanning Tree* for P is a spanning tree with no crossing edges. In this section we show how to enumerate all Euclidean spanning trees for P. First we describe an *optimum* tree T^*. By relabeling the points if necessary, we may assume that p_1 is the lexicographically smallest point, and hence an extreme point of the convex hull of P. We label the other points $p_2 \ldots p_n$ in sorted counter-clockwise order about p_1 so that both p_1p_2 and p_1p_n are edges of the convex hull of P. T^* is defined as the tree consisting of edges p_1p_i $i = 2, \ldots, n$. It is clearly an Euclidean spanning tree for P. The enumeration algorithm is based on the following lemma.

Lemma 3.15. *For any non-optimum Euclidean spanning tree T of P there is an edge e of T^* which is not in T and an edge f of T which is not in T^* such that $T' = T + e - f$ is an Euclidean spanning tree.*

Proof. The proof uses an adoption of an argument due to F.F. Yao [23]. A candidate for f is an edge p_ip_j that is in $T - T^*$. Note that this implies that neither endpoint is p_1. At least one of the edges p_1p_i and p_1p_j is not in T, and can be used for e provided it is not intersected by any other edge of T. By placing an acyclic relation on the edges of $T - T^*$ we show the existence of such a pair of edges f and e.

By convention, when we refer to an edge p_ip_j of T we will assume that $i < j$. Let p_ip_j and p_rp_s be two edges of $T - T^*$. We say that p_ip_j *dom* p_rp_s whenever edge p_ip_j intersects the interior of the triangle $p_1p_rp_s$. If p_ip_j *dom* p_rp_s and in addition p_ip_j crosses p_1p_r then we say that p_ip_j *leftdom* p_rp_s, otherwise we say that p_ip_j *rightdom* p_rp_s. Note that p_ip_j *leftdom* p_rp_s, implies that $i < r$, so we have the following.

Observation 1: The relation *leftdom* is acyclic.

Observation 2: If p_ip_j *rightdom* p_rp_s then $i \geq r$.

Proof: Since p_ip_j and p_rp_s are edges of T they can only intersect at endpoints. Since p_ip_j does not cross p_1p_r and it intersects the interior of the triangle $p_1p_rp_s$, we must have $i \geq r$ (note equality is possible).

Let p_tp_u be an edge of $T - T^*$.

Observation 3: If p_ip_j *leftdom* p_rp_s and p_rp_s *rightdom* p_tp_u then p_ip_j *dom* p_tp_u.

Proof: First note that edges p_rp_s and p_tp_u do not cross because they are edges of T. Since p_rp_s *rightdom* p_tp_u either $r = t$ or $r > t$. If $r = t$ then p_ip_j *leftdom* p_tp_u. If $r > t$ then p_r is contained in the interior of the triangle $p_1p_tp_u$. Since p_ip_j crosses p_1p_r it also properly intersects the interior of the triangle $p_1p_tp_u$, as required.

Using the above observations we can show that *dom* has a maximal element, which is our candidate for f. Since *leftdom* is acyclic, it has maximal element(s). Let p_tp_u be the maximal element of *leftdom* satisfying the conditions:

(i) if p_ip_j is a maximal element of *leftdom* then $i \leq t$, and

(ii) if p_tp_j is a maximal element of *leftdom*, $j \neq u$, then $\angle p_1p_tp_u < \angle p_1p_tp_j$.

Claim: p_tp_u is a maximal element of *dom*.

Proof: Assume, on the contrary, the existence of an edge p_rp_s such that p_rp_s *dom* p_tp_u. First note that since p_tp_u is maximal for *leftdom* we must have p_rp_s *rightdom* p_tp_u Let p_ip_j be the maximal element for *leftdom* in the chain of the *leftdom* relation containing p_rp_s. By repeated use of Observation 3 and the maximality of p_tp_u with respect to *leftdom*, we have p_ip_j *rightdom* p_tp_u. By Observation 2, we have $i \geq t$, which combined with condition (i) gives $i = t$. But now condition (ii) gives a contradiction, since we cannot have p_tp_j *rightdom* p_tp_u This contradiction proves the claim.

The rest of the proof is straightforward. We let f be the edge p_tp_u. Since T is a tree, at least one of p_1p_t, p_1p_u is not in T and is our choice for e. The maximality of f with respect to *dom* proves that T' is Euclidean. ∎

Lemma 3.15 immediately shows that there is a reverse search algorithm for enumerating all Euclidean spanning trees. The performance of an implementation is another matter. Here we do not try to search for the most efficient one, since it will go beyond the scope of this paper. Instead, we shall describe a simple one with a reasonably good time complexity.

First we define the graph $G_{EST} = (V_{EST}, E_{EST})$, where V_{EST} is the set of all Euclidean spanning trees, and two trees T and T' in V_{EST} are adjacent in G_{EST} if the symmetric difference $T \triangle T'$ consists of two edges of form, $p_1 p_j$ and $p_r p_s$ with $j = r$ or $j = s$.

For any Euclidean spanning tree T different from T^*, let $M(T)$ be the set of edges in $T \setminus T^*$ that are maximal elements of *dom*. Lemma 3.15 guarantees that $M(T)$ is nonempty. For each edge in $M(T)$, there exists a unique edge $e_f \in T^*$ such that $T - f + e_f \in V_{EST}$. The local search we use here is $(G_{EST}, S_{EST}, f_{EST})$ where $S_{EST} = \{T^*\}$ and

$$f_{EST}(T) := T - f + e_f$$

where f is the lexico-min edge in $M(T)$.

In order to construct an A-oracle, one can set δ_{EST} to be $(n-1)(n-2)$ since there are $(n-1)$ edge candidates to remove from T and at most $(n-2)$ edges to add, at any Euclidean spanning tree T. We can easily implement Adj_{EST} such that $t(Adj_{EST}) = O(n)$ since recognizing whether two edges cross takes only constant time. Similarly we have $t(f_{EST}) = O(n)$.

Theorem 3.16. *There is an implementation of ReverseSearch(Adj_{EST}, δ_{EST}, S_{EST}, f_{EST}) for the Euclidean spanning tree enumeration problem with time complexity $O(n^3\,|V_{EST}|)$ and space complexity $O(n)$.*

Proof. To prove this, we simply use Corollary 2.3 which claims the time complexity of ReverseSearch is $O(\delta(t(Adj) + t(f))|V|)$. This together with the discussion above gives the time complexity. The space complexity is obvious. ∎

4 Partial Reverse Search

In Section 2, we developed the reverse search as a general exhaustive search technique. Here we introduce a simple modification of reverse search as a general algorithmic framework for solving a certain class of hard optimization problems.

In order to understand the main idea, let us first consider the 0-1-integer programming problem (abbreviated by *IP*):

$$\begin{array}{ll} \text{maximize} & cx \\ \text{subject to} & Ax \leq b \text{ and}, \\ & x_j = 0 \text{ or } 1 \text{ for } j = 1, \ldots, n, \end{array} \tag{4.2}$$

where A is a rational $m \times n$ matrix, b a rational m-vector and c a rational n-vector. It is well known that IP is NP-complete [19]. A standard technique used to solve this problem is the branch-and-bound (see, e.g. Chapter 18 in [16]). The new algorithm to be introduced now can be considered as an alternative to the branch-and-bound methods.

Let

$$P = \{x : Ax \leq b \text{ and } 0 \leq x \leq 1\}.$$

Then we have the following lemma.

Lemma 4.1. *A point x in P is an optimal solution to the IP (4.2) if and only if x maximizes cx over all vertices of P that are integral.*

Proof. The "only if" part is trivial. The "if" part follows from the fact that every integral vector of P is a vertex of P. ∎

This lemma suggests the following primitive strategy to solve an IP:

(1) Apply reverse search of Section 3.1 for enumerating all vertices of P;

(2) during the search procedure, output no vertices but remember and update an integral vertex, say $\overline{v}$ with currently best objective value; and

(3) output $\overline{v}$ at the end of search.

This procedure obviously works. But it is very far from practical since the vertex enumeration takes too much time for large n and m. Can we shorten this procedure and somehow overcome this difficulty? Observe that the local search f we reverse in (1) is the simplex method, and f is *monotone* with respect to the objective function, that is, $cx_B \leq cx_{f(B)}$, where x_B denotes the basic solution (i.e. a vertex of P) associated with a basis B. This means that as we follow the trace T_f reversely (against its orientation), the objective value monotonically decreases. Therefore, while doing reverse search, as soon as we detect an integral vertex or a vertex with objective value worse than the current best value $c\overline{v}$, there is no reason to search lower in the trace.

We do not know if this "partial reverse search" strategy yields a practical algorithm. But we believe that it deserves further investigation. Unlike the branch-and-bound algorithms for IP that require a large number of linear programs to be solved, we have to solve at most one linear program initially. Furthermore we have simple ways to implement it in parallel computers.

Now, let us present the partial reverse search in general setting. Suppose we have a finite local search (G, S, f) given by an A-oracle. Therefore, we can apply reverse search to enumerate all vertices of $G = (V, E)$. Now, in addition, suppose we have the following situation:

(1) we are given an objective function c defined at each vertex;

(2) we are given a boolean function Q defined at each vertex.

The general problem to be solved is to

$$\begin{array}{ll} \text{maximize} & c(v) \\ \text{subject to} & v \in V \text{ and}, \\ & Q(v) = \textbf{true}. \end{array} \tag{4.3}$$

Obviously, an IP is a special case of this optimization problem. We call the following procedure a *partial reverse search*:

```
procedure PartialReverseSearch(Adj,δ,S,f,c,Q);
  for each vertex s ∈ S do
    v := s; j := 0; (* j: neighbor counter *)
    v̄ := 0; c̄ = −∞; (* current best solution and value *)
    repeat
      while j < δ do
        j := j + 1;
        next := Adj(v, j);
        if next ≠ 0 and f(next) = v and c(next) > c̄ then  (* reverse traverse *)
          v := next; j := 0;
          if Q(v) = true then  (* update the current best solution *)
            v̄ := v; c̄ = c(v);
            j := δ + 1 (* no further reverse *)
          endif
        endif
      endwhile;
      if v ≠ s then (* forward traverse *)
        u := v;    v := f(v);
        determine j such that Adj(v, j) = u (* restore j *)
      endif
    until v = s and j = δ;
    output v̄
  endfor.
```

We call a local search function f *monotone with respect to* c if $c(v) \le c(f(v))$ for all $v \in V \setminus S$. Then, the previous discussion of partial reverse algorithm for IP naturally extends to:

Proposition 4.2. *The partial reverse search solves the optimization problem (4.3) if the function f is monotone with respect to c.*

One can easily find applications of partial reverse search other than the integer programming. For example, one can find a very special triangulation of points in the plane by using the flip algorithm f_{TRI} of Section 3.3. It is often desired to have a triangulation which does not use a very narrow angle in any of its triangles. In fact, a Delaunay triangulation is one that maximizes the angle vector, and thus in particular, it maximizes the minimum angle. The partial reverse search can then find a triangulation satisfying any prescribed condition(s) Q and maximizing the angle vector.

Another example is to find a "special" basis of a weighted matroid (or a spanning tree of a graph with weighted edges). It is easy to modify the local search f_{BAS} of Section 3.6 so that it finds a basis of maximum weight (see, [9]). Setting Q to be any condition(s) that you want a basis to satisfy, e.g. having k leaves for the graph case. Then the partial reverse search finds a basis satisfying Q with maximum weight.

Again, we have no evidence whatsoever supporting these applications being useful in practice. In order to say anything meaningful in this respect demands further research.

Acknowledgements. The authors are grateful for Département de Mathematiques of EPFL, Switzerland, where the major part of this paper was written during the second author's visit in autumn 1991. In particular, the idea of using the flip algorithm for the triangulation enumeration came from the author's discussion with Professor Th. M. Liebling of EPFL.

References

[1] A.V. Aho, J.E. Hopcroft an J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1987.

[2] D. Avis, "A C implementation of the reverse search vertex enumeration algorithm," Research Report B-359, Department of Information Sciences, Tokyo Institute of Technology (1992).

[3] D. Avis and V. Chvátal, "Notes on Bland's pivoting rule," *Mathematical Programming* 8, (1978), 24-34.

[4] D. Avis and K. Fukuda, "A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra," *Discrete Comput Geometry* 8 (1992), 295-313.

[5] D. Avis and K. Fukuda, "A basis enumeration algorithm for linear systems with geometric applications," *Applied Mathematics Letters* 5 (1991), 39-42.

[6] R.C. Buck, "Partion of space," *Amer. Math. Monthly* 50 (1943), 541-544.

[7] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.

[8] S. Fortune, "A note on Delaunay diagonal flips," AT&T Bell Laboratories, Murray Hill, New Jersey, 1987.

[9] S. Fujishige, *Submodular Functions and Optimization*, Annals of Discrete Mathematics 47, North-Holland, 1991.

[10] K. Fukuda, K. Saito and A. Tamura, "Combinatorical face enumeration in arrangements and oriented matroids," *Discrete Appl. Math.* 31 (1991), 141-149.

[11] K. Fukuda, K. Saito, A. Tamura and T.Tokuyama "Bounding the number of k-faces in arrangements of hyperplanes," *Discrete Appl. Math.* 31 (1991), 151-165.

[12] K. Fukuda and I. Mizukoshi, Mathematica package: Vertex enumeration for convex polyhedra and hyperplane arrangements, Version 0.3 Beta, Graduate School of Systems Management, University of Tsukuba, Tokyo (1991), available via anonymous ftp from cs.sunysb.edu (directory pub/Combinatorica).

[13] L.J. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams," *ACM Trans. Graphics* 4 (1985), 74-123.

[14] D.E. Knuth and J.L. Szwarcfiter , "A structured program to generate all topological sorting arrangements," *Information Processing Letters* 2 (1974), 153-157.

[15] T. Matsui, "Algorithms for finding all the spanning trees in undirected graphs," METR93-08, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, (1993).

[16] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*, Printice-Hall, 1982.

[17] G. Pruesse and F. Ruskey "Generating Linear Extensions Fast," *SIAM J. Computing*, to appear.

[18] R.C Read and R.E. Tarjan, "Bounds on backtrack algorithms for listing cycles, paths, and spanning trees," *Networks*, 5 (1975), 237-252.

[19] A. Schrijver, *Theory of Linear and Integer programming*, John Wiley & Sons, 1986.

[20] A. Shioura and A. Tamura, "Efficiently scanning all spanning trees of an undirected graph," Research Report B-270, Dept. of Information Sciences, Tokyo Institute of Technology (1993).

[21] H. Telley, "Static and dynamic weighted Delaunay triangulations in the Euclidean plane and in the flat torus," Research Report No. UIUCDCS-R-90-1662, Department of Computer Science, University of Illinois at Urbana-Champaign (1990).

[22] D.J.A. Welsh, *Matroid Theory*, Academic Press, 1976.

[23] F.F. Yao, "On the priority approach to hidden surface algorithms," Proc. 21st Symposium on Foundations of Computer Science (1980), 301-307.

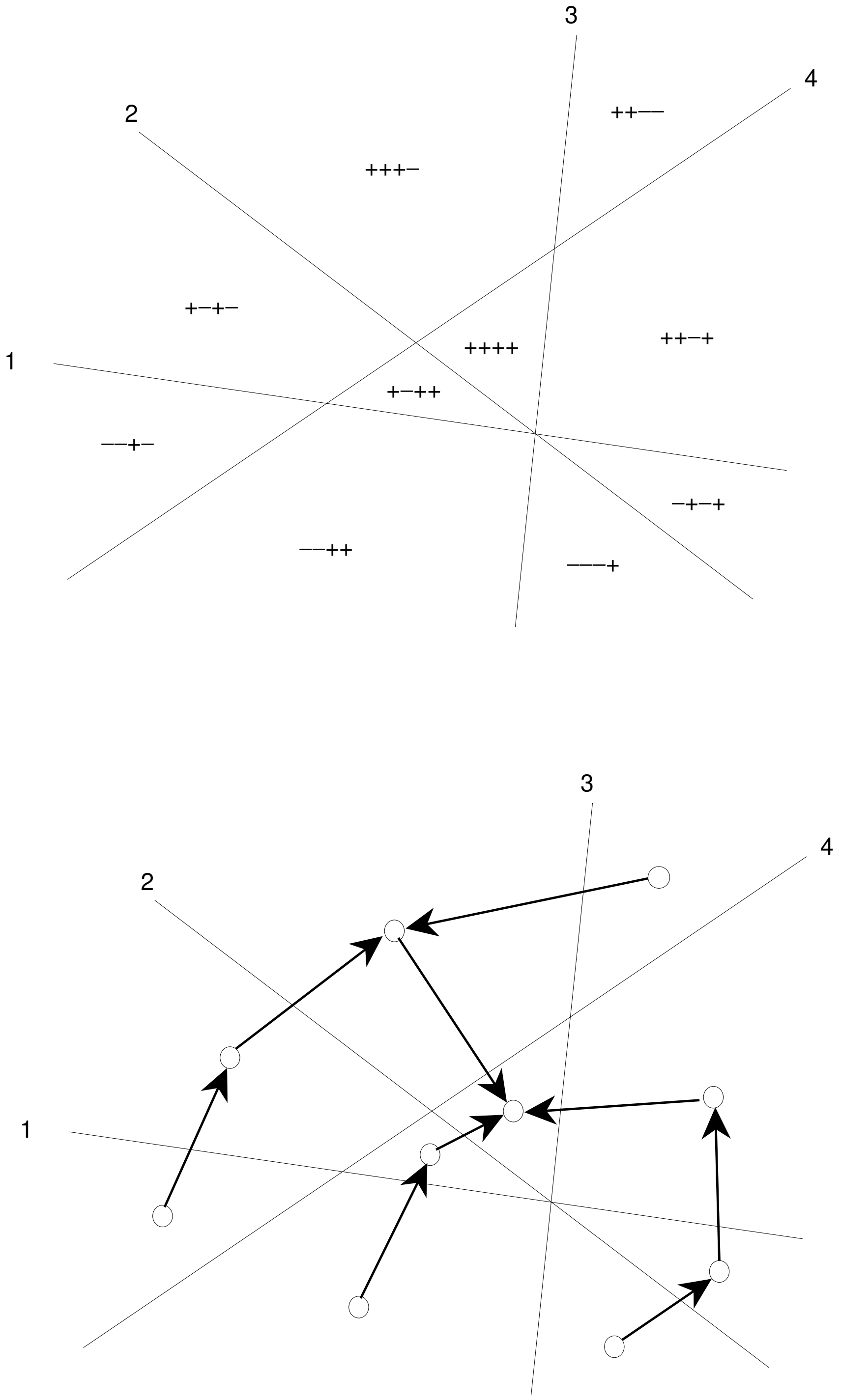

Figure 4.1: An arrangement of hyperplanes and the trace of f_{CELL}

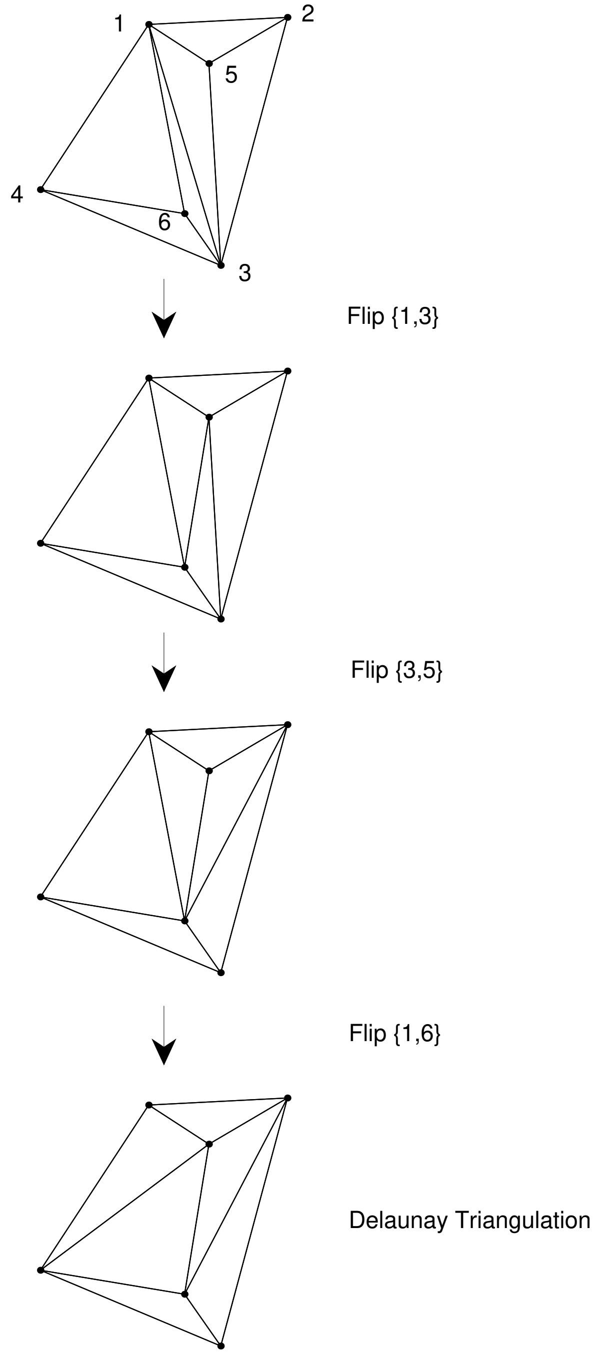

Figure 4.2: Flip Algorithm and Delaunay Triangulation

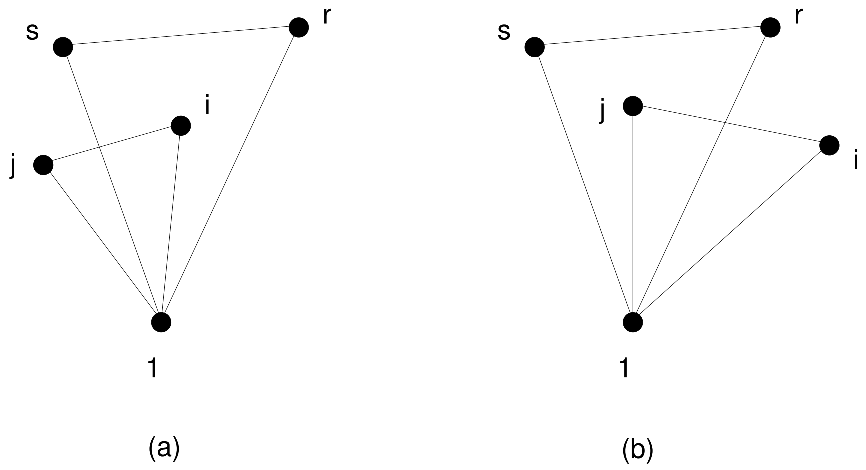

Figure 4.3: (a) $p_i p_j$ *rightdom* $p_r p_s$ (b) $p_i p_j$ *leftdom* $p_r p_s$