Computational Geometry: Methods and Applications

Jianer Chen

Department of Computer Science & Engineering
Texas A&M University

January 12, 2023

Chapter 1

Introduction

Geometric objects such as points, lines, and polygons are the basis of a broad variety of important applications and give rise to an interesting set of problems and algorithms. The name geometry reminds us of its earliest use: for the measurement of land and materials. Today, computers are being extensively to solve larger-scale geometric problems. Over the past few decades, a set of tools and techniques has been developed that takes advantage of the structure provided by geometry. This discipline is known as *Computational Geometry.*

The discipline was named and largely started around later part of 1970's by Shamos, whose Ph.D. thesis attracted considerable attention. After a decade of development the field came into its own in 1980s, when three components of any healthy discipline were realized: a textbook, a conference, and a journal. Preparata and Shamos's book *Computational Geometry: An Introduction* [23], the first textbook solely devoted to the topic, was published at about the same time as the first ACM Symposium on Computational Geometry was held, and just prior to the start of a new Springer-Verlag journal *Discrete and Computational Geometry.* The field is currently thriving. Since 1985, several texts, collections, and monographs have appeared [1, 10, 18, 20, 26, 27]. The annual symposium has attracted over 100 papers and several hundreds of attendees steadily. There is evidence that the field is broadening to touch geometric modeling and geometric theorem proving. Perhaps most importantly, students who obtained their Ph.D.s in computer science with theses in computational geometry have graduated, obtained positions, and are now training the next generation of researchers.

Computational geometry is of practical importance because Euclidean space of two and three dimensions forms the arena in which real physical

objects are arranged. A large number of application areas such as pattern recognition [29], computer graphics [19], image processing [22], operations research, statistics [4, 28], computer-aided design, robotics [26, 27], etc., have been the incubation bed of the discipline since they provide inherently geometric problems for which efficient algorithms have to be developed. A large number of manufacturing problems involve wire layout, facilities location, cutting-stock and related geometric optimization problems. Solving these efficiently on a high-speed computer requires the development of new geometrical tools, as well as the application of fast-algorithm techniques, and is not simply a matter of translating well-known theorems into computer programs. From a theoretical standpoint, the complexity of geometric algorithms is of interest because it sheds new light on the intrinsic difficulty of computation.

In this book, we concentrate on four major directions in computational geometry: the construction of convex hulls, proximity problems, searching problems and intersection problems.

Chapter 2

Algorithmic Foundations

The analysis and design of computer algorithms has been one of the most thriving endeavors in computer science. The fundamental works of Knuth [14] and Aho-Hopcroft-Ullman [2] have brought order and systematization to a rich collection of isolated results, conceptualized the basic paradigms, and established a methodology that has become the standard of the field. It is beyond the scope of this book to review in detail the material of those excellent texts, with which the reader is assumed to be reasonably familiar. It is appropriate, however, at least from the point of view of terminology, to briefly review the basic components of the language in which computational geometry will be described. These components are algorithms and data structures. Algorithms are programs to be executed on a suitable abstraction of actual "von Neumann" computers; data structures are ways to organize information, which, in conjunction with algorithms, permit the efficient and elegant solution of computational problems.

2.1 The computational model

Many formal models of computation appear in the literature. There is no general consensus as to which of these is the best. In this book, we will adopt the most commonly-used model. More specifically, we will adopt random access machines (RAM) as our computational model.

Random access machine (RAM)

A random access machine (RAM) models a single-processor computer with a random access memory.

A RAM consists of a read-only input tape, a write-only output tape, a program and a (random access) memory. The memory consists of registers each capable of holding a real number of arbitrary precision. There is also no upper bound on the memory size. All computations take place in the processor. A RAM can access (read or write) any register in the memory in one time unit when it has the correct address of that register.

The following operations on real numbers can be done in unit time by a random access machine :

1) Arithmetic operations: $*$, /, $+$, $-$, log, exp, sin.
2) Comparisons
3) Indirect access

2.2 Complexity of algorithms and problems

The following notations have become standard:

- $O(f(n))$: the class C_1 of functions such that for any $g \in C_1$, there is a constant c_g such that $f(n) \geq c_g g(n)$ for all but a finite number of n's. Roughly speaking, $O(f(n))$ is the class of functions that are at most as large as $f(n)$.

- $o(f(n))$: the class C_2 of functions such that for any $g \in C_2$, $\lim_{n\to\infty} g(n)/f(n) = 0$. Roughly speaking, $o(f(n))$ is the class of functions that are asymptotically smaller than $f(n)$.

- $\Omega(f(n))$: the class C_3 of functions such that for any $g \in C_3$, there is a constant c_g such that $f(n) \leq c_g g(n)$ for all but a finite number of n's. Roughly speaking, $\Omega(f(n))$ is the class of functions which are at least as large as $f(n)$.

- $\omega(f(n))$: the class C_4 of functions such that for any $g \in C_4$, $\lim_{n\to\infty} f(n)/g(n) = 0$. Roughly speaking, $\omega(f(n))$ is the class of functions that are asymptotically larger than $f(n)$.

- $\Theta(f(n))$: the class C_5 of functions such that for any $g \in C_5$, $g(n) = O(f(n)$ and $g(n) = \Omega(f(n))$. Roughly speaking, $\Theta(f(n))$ is the class of functions which are of the same order as $f(n)$.

Complexity of algorithms

Let $\mathcal{A}$ be an algorithm implemented on a RAM. If for an input of size n, $\mathcal{A}$ halts after m steps, we say that the running time of the algorithm $\mathcal{A}$ is m

on that input.

There are two types of analyses of algorithms: worst case and expected case. For the worst case analysis, we seek the maximum amount of time used by the algorithm for all possible inputs. For the expected case analysis we normally assume a certain probabilistic distribution on the input and study the performance of the algorithm for any input drawn from the distribution. Mostly, we are interested in the asymptotic analysis, i.e., the behavior of the algorithm as the input size approaches infinity. Since expected case analysis is usually harder to tackle, and moreover the probabilistic assumption sometimes is difficult to justify, emphasis will be placed on the worst case analysis. Unless otherwise specified, we shall consider only worst case analysis.

Definition Let $\mathcal{A}$ be an algorithm. The *time complexity* of $\mathcal{A}$ is $O(f(n))$ if there exists a constant c such that for every integer $n \geq 0$, the running time of $\mathcal{A}$ is at most $c \cdot f(n)$ for all inputs of size n.

Complexity of problems

While time complexity for an algorithm is fixed, this is not so for problems. For example, Sorting can be implemented by algorithms of different time complexity. The time complexity of a known algorithm for a problem gives us the information about *at most* how much time we need to solve the problem. We would also like to know the *minimum* amount of time we need to solve the problem.

Definition A function $u(n)$ is an *upper bound* on the time complexity of a problem $\mathcal{P}$ if there is an algorithm $\mathcal{A}$ solving $\mathcal{P}$ such that the running time of $\mathcal{A}$ is $u(n)$. A function $l(n)$ is a *lower bound* on the time complexity of a problem $\mathcal{P}$ if *any* algorithm solving $\mathcal{P}$ has time complexity at least $l(n)$.

2.3 A data structure supporting set operations

A *set* is a collection of *elements*. All elements of a set are different, which means no set can contain two copies of the same element.

When used as tools in computational geometry, elements of a set usually are normal geometric objects, such as points, straight lines, line segments, and planes in Euclidean spaces.

We shall sometimes assume that elements of a set are linearly ordered by a relation, usually denoted "$<$" and read "less than" or "precedes". For example, we can order a set of points in the 2-dimensional Euclidean space by their x-coordinates.

Let S be a set and let u be an arbitrary element of a universal set of which S is a subset. The fundamental operations occurring in set manipulation include:

- Search(u, S): Is $u \in S$?
- Insert(u, S): Add the element u to the set S.
- Delete(u, S): Remove the element u from the set S.

When the universal set is linearly ordered, the following operations are also important:

- Min(S): Report the minimum element of the set S.
- Split(u, S): Partition the set S into two sets S_1 and S_2, so that S_1 contains all the elements of S that are smaller than or equal to u, and S_2 contains all the elements of S that are larger than u.
- Splice(S, S_1, S_2): Assuming that all elements in the set S_1 are smaller than any element in the set S_2, form the ordered set $S = S_1 \cup S_2$.

We will introduce a special data structure: 2-3 trees, which represent sets of elements and support the above set operations efficiently.

Definition A *2-3 tree* is a tree such that each non-leaf node has two or three children, and every path from the root to a leaf is of the same length.

The following theorem can be proved using induction on n, and the proof is left to the reader.

Theorem 2.3.1 *A 2-3 tree of n leaves has height bounded by* $\log n$.

A linearly ordered set of elements can be represented by a 2-3 tree by assigning the elements to the leaves of the tree in such a way that for any non-leaf node v of the tree, all elements stored in the first child of v are less than any elements stored in the second child of v, and all elements stored in the second child of v are less than any elements stored in the third child of v (if v has a third child).

Each non-leaf node v of a 2-3 tree has two or three children, which will be named child1(v), child2(v), child3(v), respectively. The node v also keeps three values for the corresponding subtrees:

- $l(v)$: the largest element stored in the subtree rooted at child1(v).
- $m(v)$: the largest element stored in the subtree rooted at child2(v).
- $h(v)$: the largest element stored in the subtree rooted at its child3(v) (if child3(v) exists).

Remark. Strictly speaking, the third value $h(v)$ is not needed. All algorithms can be implemented without the value $h(v)$, and without increasing the time complexity. However, we suggest to keep the third value in an implementation, which will simplify certain implementation details.

2.3.1 Searching

The algorithm to search an element in a 2-3 tree is given as follows, where r is the root of the 2-3 tree, and x is the element to be searched in the tree.

```
Algorithm  Search(r, x)
1. If (r is empty) return "NO";
2. If (r is a leaf node) return (value(r) == x);
3. If (l(r) >= x) return Search(child1(r), x);
   Else If (m(r) >= x) return Search(child2(r), x);
   Else If (r has a third child) return Search(child3(r), x);
   Else return "NO".
```

Since the height of a 2-3 tree is $O(\log n)$, and the algorithm simply follows a path in the tree from the root to a leaf, and spends time $O(1)$ on each level, the time complexity of the algorithm `Search` is $O(\log n)$, where n is the number of leaves in the tree.

2.3.2 Minimum and Maximum

Given a 2-3 tree T we want to find out the minimum element stored in the tree. Recall that in a 2-3 tree the elements are stored in leaf nodes in *ascending* order from left to right. Therefore the problem is reduced to going down the tree, always selecting the left most link, until a leaf node is reached. This leaf node should contain the minimum element stored in the tree. Evidently, the time complexity of this algorithm is $O(\log n)$ for a 2-3 tree with n leaves.

```
Algorithm  Min(r)
1.  If (r is empty) return failure;
2.  If (r is a leaf) return value(r);
    Else return Min(child1(r)).
```

Similarly, the maximum element stored in a 2-3 tree can be found in time $O(\log n)$.

2.3.3 Insertion

To insert a new element x into a 2-3 tree T rooted at r, we apply a recursive algorithm that dose two things: (1) insert x into the tree T rooted at r; and (2) report whether this insertion splits the tree T rooted at r into two 2-3 trees.

If the 2-3 tree T has at most one leaf, then the job is easy: (1) if T has no leaf (i.e., T represents an empty set), then we simply make a 2-3 tree that consists of a single node, which is both the root and the leaf of the tree, with a value x. (2) if T has only one leaf of value y, then the tree T is a single-node tree, inserting x into T makes a two-leaf tree, whose values are x and y, respectively, and the leaves are ordered properly.

Now suppose that the 2-3 tree T has a height at least 1 with at least two leaves, then we proceed at first as if we were searching x in the tree T. However, at the level just above the leaves, we start our insertion operation recursively. In general, suppose that we want to add a new child w to a node v in the 2-3 tree T. If v has only two children, we simply make w a new child of v, placing the children in the proper order and updating the information of the node v.

Suppose, however, that v already has three children v_1, v_2, and v_3. Then w would be the fourth child of v. We cannot have a node with four children in a 2-3 tree, so we split the node v into two nodes, which we call v and v'. With the new node v', we can let the first two of $\{v_1, v_2, v_3, w\}$ (in terms of the linear order) be children of v, and let the rest two be children of v'. Now, the node v' is the root of a subtree and should be added as a new child to the parent of v. Thus, the operation now can be recursively done at the level of the parent of v.

One special case occurs when we wind up splitting the root. In that case we create a new root, whose two children are the two nodes into which the old root was split. This is how the number of levels in (i.e., the height of) a 2-3 tree increases.

The above discussion is implemented as the following algorithms, where r is the root of the 2-3 tree to which the element x is to be inserted.

```
Algorithm  Insert(r, x)
1. If (the tree rooted at r has < 2 leaves)
   process directly; return;
2. AddLeaf(r, x, r');
3. If (r' != NULL)
   create a new node v; let r and r' be children of v;  r = v.
```

The procedure `AddLeaf(r,x,r')` above is implemented by the following recursive algorithm, which inserts a new element x to the 2-3 tree rooted at r. Moreover, if this insertion causes splitting the node r due to exceeding the number of children, then a new node r' is created to take two children from r. Therefore, if r' is not empty when the procedure returns, then r and r', respectively, are the roots of two 2-3 trees of the same height.

```
Algorithm  AddLeaf(r, x, r')  /* the node r is not a leaf */
1. r' = NULL;
2. If (r is a parent of leaves)
      If (r has 2 children) add x as a new child of r;
      Else /* r has 3 children */
        order x and the three children of r in the linear order;
        let the first two be children of r; and the rest two be children of r';
      return;
3. If (l(r) >= x) v = child1(r);
   Else If (m(r) >= x or child3(r)==NULL) v = child2(r);
   Else v = child3(r);
4. AddLeaf(v, x, v');
5. If (v' == Null) return;
6. If (v' != NULL and r has 2 children) add v' as a new child of r;
   Else  /* r has 3 children and v' is not NULL */
     order v' and the three children of r in the linear order;
     let the first two be children of r; and the rest two be children of r';
     return.
```

Analysis: Clearly, the running time of the algorithm `Insert` is dominated by that of the procedure `AddLeaf`, which at each level of the 2-3 tree spends constant time (see steps 1-3, 5-6 of the procedure `Addson`). Since a 2-3 tree with n leaves has a height bounded by $\log n$, we conclude that the algorithm `Insert` runs in time $O(\log n)$.

2.3.4 Deletion

When we delete a leaf from a 2-3 tree, we may leave its parent v with only one child. If v is the root, delete v and let its lone child be the new root. Otherwise, let p be the parent of v. If p has another child, adjacent to v on either the right or the left, and that child of p has three children, we can transfer the proper one of those three to v. Then v has two children, and we are done.

If the children of p adjacent to v have only two children, transfer the lone child of v to an adjacent sibling of v, and delete v. Should p now have only one child, repeat all the above, recursively, with p in place of v.

Summarizing these discussions together, we get the algorithm `Delete`, as shown below, where procedure `Delete`() is merely a driver for sub-procedure `Del`() in which the actual work is done.

The variables `done` and `1son` in `Del`() are boolean flags used to indicate successful deletion and to detect the case when a node in the tree has only one child, respectively.

In the worst case we need to traverse a path in the tree from root to a leaf to locate the leaf to be deleted, then from that leaf node to the root, in case that every non-leaf node on the path has only two children in the original 2-3 tree T. Thus the time complexity of `Delete` algorithm for a 2-3 tree with n leaves is $O(\log n)$.

```
Algorithm  Delete(r, x)
1. If (r == Null) return failure;
2. If (r is a leaf)
     If (x == l(r)) r = Null; return;
     Else return failure;
3. Del(r, x, done, 1son);
4. If (done == false) return failure;
5. If (1son == true) r = child1(r); return.

Algorithm  Del(r, x, done, 1son)
1. done = true; 1son = false;
2. If  (r is a parent of leaves)  process properly and return;
    /* i.e., delete x if it is in the tree; update done and 1son */
3. If (x <= l(r))  r' = child1(r);
   Else if (x <= m(r)) or (child3(r) == Null)  r' = child2(r);
   Else  r' = child3(r);
4. Del(r', x, done', 1son');
5. If (done' == false)  done = false; return;
6. If (1son' == true)
     If (r has at least 4 grandchildren)
       reorganize the grandchildren of r so that each of r and its
       children has either 2 or 3 children; return;
     Else make r a 1-child node (with 3 grandchildren);
       1son = true; return.
```

2.3.5 Splice

Splicing two trees into one big tree is a special case of the more general operation of merging two trees. Splice assumes that all the keys in one of the trees are larger than all those in the other tree. This assumption effectively reduces the problem of merging the trees into "pasting" the shorter tree into a proper position in the taller tree. "Pasting" the shorter tree is actually no

more than performing an `AddLeaf` operation to a proper node in the taller tree.

To be more specific, let T_1 and T_2 be two 2-3 trees which we wish to splice into a single 2-3 tree T, where all keys in T_1 are smaller than that in T_2. Furthermore, assume that the height of T_1 is less than or equal to that of T_2 so that T_1 is "pasted" to T_2 as a left child of a leftmost node at the proper level in T_2. In the case where the heights are equal, the new tree T can be easily constructed by letting T_1 and T_2 be the two children of the root of T. Otherwise, a node v at a proper level in the tree T_2 is found, and T_1 is inserted as the left child of v. Note that the level of the node v in the tree T_2 is given by (assume the root of T_2 is at level 0):

$$height(T_2) - height(T_1) - 1$$

A more detailed description of the algorithm `Splice` is given as follows.

```
Algorithm Splice(T, T1, T2)
/* Assume  all elements in T1 are less than any elements in T2 */
1. h1 = height of T1;  h2 = height of T2;
2. If h1 == h2
    create a root r for T and let T1 and T2 be children of r; return;
3. If h1 < h2  find the leftmost node v in T2 at level h2-h1-1,
    add T1 as a new child of v;  T = T2;  return;
4. If h1 > h2  find the rightmost node v in T1 at level h1-h2-1,
    add T2 as a new child of v;  T = T1;  return.
```

Note that steps 3-4 in the algorithm `Splice` may cause nodes in a 2-3 tree with more than 3 children. Therefore, these steps should really be implemented as recursive procedures that are similar to the algorithm `AddLeaf` as gaven in the last subsection. However, instead of stopping at the level of nodes that are parents of leaves, here the recursions stop when the height of the taller tree is equal to 1 plus the height of the shorter tree.

The heights `h1` and `h2` of the trees `T1` and `T2`, respectively, in step 1 can be computed by tracing a path in the trees from the root to (any) leaf. Thus, step 1 takes time $O(\log n)$. So the algorithm `Splice` runs in time $O(\log n)$. If we already know the values of `h1` and `h2` so step 1 of the algorithm can be omitted, then the algorithm follows a path in the taller tree from the root to a node a level h, where h is the difference of the heights of the two trees `T1` and `T2` minus 1. Thus, under this assumption, the running time of the algorithm `Splice` will be $O(h)$. We summarize the discussion in the following theorem.

Theorem 2.3.2 *The algorithm* `Splice` *takes time* $O(\log n)$. *If the heights of the two trees are known, then the two trees can be spliced in time* $O(h)$, *where* h *is the difference of the heights of the two trees.*

2.3.6 Split

By splitting a given 2-3 tree T into two 2-3 trees, T_1 and T_2, at a given element x, we mean to split the tree T in such a way that all elements in T that are less than or equal to x go to T_1 while the remaining elements in T go to T_2.

The idea is as follows: based on the way we search the element x in the tree T, we in addition use two stacks to store, respectively, the subtrees to the left and the subtrees to the right of the traversed path (splitting path). Finally, the subtrees in each stack are spliced together to form the desired trees T_1 and T_2. The algorithm is given as follows.

```
Algorithm  Split(T, x, T1, T2)
/* Split T into T1 and T2 such that all elements in T1 are <= x, and all
   elements in T2 are > x, where SL and SR are stacks.*/
1. let r be the root of T;
2. While r is not a leaf Do
     If (x <= l(r))
       If (r has a third child) SR <-- child3(r);
       SR <-- child2(r);
       r = child1(r);
     Else If l(r) < x <= m(r)
       SL <-- child1(r);
       If (r has a third child) SR <-- child3(r);
       r = child2(r);
     Else /* x is in the third child of r */
       SL <-- child1(r); SL <-- child2(r);
       r = child3(r);
/* construct T1 */
3. T1 <-- SL;
4. While SL is not empty Do
     t <-- SL;
     Splice(T1, t, T1);
/* construct T2 */
5. T2 <-- SR;
6. While SR  is not empty Do
     t <-- SR;
     Splice(T2, T2, t);
```

Note that we have omitted certain special cases in the above algorithm. For example, if `x` is smaller than all elements in `T`, then we would have `T1` $= \emptyset$ and `T2` $=$ `T`. Similarly we can handle the case where `x` is larger than all elements in `T`. These cases can be tested and processed in time $O(\log n)$.

Suppose that the subtrees in the stack `SL` are τ_1, τ_2, ..., τ_h, which were pushed into the stack `SL` in this order. By the properties of a 2-3 tree, we know that for all i, all elements in the subtree τ_i are smaller than any element in the subtree τ_{i-1}. Since the subtrees in `SL` are popped out from `SL` in the order of τ_h, ..., τ_2, τ_1 and are spliced in the tree `T1` (steps 3-

4), we know that the splice operation `Splice(T1, t, T1)` is always valid. Similarly, steps 5-6 are valid.

It is easy to see that the `While` loop in step 2 takes time $O(\log n)$. The analysis for the rest of the algorithm is a bit more complicated. In each of steps 3-4 and steps 5-6, we may need to splice more than a constant number of subtrees. Thus, if we count the complexity of each splice as $O(\log n)$, we would not be able to bound the running time of these steps by $O(\log n)$.

Note that the heights of the subtrees in the stacks `SL` and `SR` can be easily computed while we traverse the path in `T` from its root in step 2 of the algorithm `Split`. By taking advantage of this fact and Theorem 2.3.2, we can have more precise analysis for the complexity of the algorithm `Split`.

The use of the stacks `SL` and `SR` to store the subtrees guarantees that the height of a subtree closer to a stack top is less than or equal to the height of the subtree immediately deeper in the stack. A crucial observation is that since we splice shorter trees first (which are on the top part of the stacks), the difference between the heights of two trees to be spliced is always very small. In fact, the total time spent on splicing all these subtrees is bounded by $O(\log n)$. We give a formal proof as follows.

Assume before we start step 4, the subtrees stored in the stack `SL` are

$$\tau_1, \tau_2, \cdots, \tau_r, \tag{2.1}$$

in the order from the top to the bottom in the stack `SL`. For a 2-3 tree τ, denote by $ht(\tau)$ the height of τ. According to the algorithm `Split`, we have

$$ht(\tau_1) \leq ht(\tau_2) \leq \cdots \leq ht(\tau_r)$$

and no three consecutive subtrees in the stack have the same height. Thus, we can partition the sequence (3.1) into non-empty "segments" such that each segment contains subtrees of the same height in the sequence:

$$s_1, s_2, \cdots, s_q$$

Each s_i either is a single subtree or consists of two consecutive subtrees of the same height in sequence (3.1). Moreover, $q \leq \log n$. Let $ht(s_i)$ be the height of the subtrees contained in the segment s_i. We have

$$ht(s_1) < ht(s_2) < \cdots < ht(s_q) \tag{2.2}$$

The `While` loop in Step 4 first splices the subtrees in segment s_1 into a single 2-3 tree $T_1^{(1)}$, then recursively splices the 2-3 tree $T_1^{(i-1)}$ and the subtrees in segment s_i into a 2-3 tree $T_1^{(i)}$, for $i = 2, \ldots, q$. We have the following lemma.

Lemma 2.3.3 *For all* $2 \leq i \leq q$, $ht(s_{i-1}) \leq ht(T_1^{(i-1)}) \leq ht(s_i)$.

PROOF. The inequality $ht(s_1) \leq ht(T_1^{(1)})$ is obvious since $T_1^{(1)}$ is obtained by splicing subtrees in the segment s_1. For $i > 2$, since $T_1^{(i-1)}$ is obtained by splicing the tree $T_1^{(i-2)}$ and the subtrees in s_{i-1}, and the subtrees in s_{i-1} have height $ht(s_{i-1})$. Thus, we must have $ht(s_{i-1}) \leq ht(T_1^{(i-1)})$.

Now consider the second inequality. The 2-3 tree $T_1^{(1)}$ is obtained by splicing the subtrees in the segment s_1, which contains at most two subtrees, both of height $ht(s_1)$. Thus, the height of the 2-3 tree $T_1^{(1)}$ is at most $ht(s_1)+1$, which, by (2.2), is not larger than $ht(s_2)$. Thus, $ht(T_1^{(1)}) \leq ht(s_2)$, and the second inequality in the lemma holds true for the case $i = 2$.

Now for the case $i > 2$, consider the height $ht(T_1^{(i-1)})$ of the 2-3 tree $T_1^{(i-1)}$. The tree $T_1^{(i-1)}$ is obtained by splicing the tree $T_1^{(i-2)}$ (note $i > 2$) and the subtrees in the segment s_{i-1}. By the inductive hypothesis, $ht(T_1^{(i-2)}) \leq ht(s_{i-1})$. If the segment s_{i-1} consists of a single subtree τ of height $ht(s_{i-1})$, then splicing the tree $T_1^{(i-2)}$ of height at most $ht(s_{i-1})$ and the tree τ of height $ht(s_{i-1})$ results in a 2-3 tree $T_1^{(i-1)}$ of height at most $ht(s_{i-1}) + 1$, which, by (2.2), is not larger than $ht(s_i)$.

Now suppose that the segment s_{i-1} consists of two subtrees τ' and τ'' of height $ht(s_{i-1})$, and that $T_1^{(i-2)}$ is first spliced with τ' to result in a tree τ^+, then τ^+ is spliced with τ'' to result in the tree $T_1^{(i-1)}$. The tree τ^+ can have a height either $ht(s_{i-1})$ or $ht(s_{i-1}) + 1$ (note $ht(T_1^{(i-2)}) \leq ht(s_{i-1})$). If the height of τ^+ is $ht(s_{i-1})$, then splicing τ^+ of height $ht(s_{i-1})$ and the tree τ'' (also of height $ht(s_{i-1})$) results in the tree $T_1^{(i-1)}$ of height at most $ht(s_{i-1}) + 1 \leq ht(s_i)$. If the height of the tree τ^+ is $ht(s_{i-1}) + 1$, then the root of the tree τ^+ must have only two children (see algorithm `Insert`, step 3). Thus, splicing τ^+ and τ'' will not increase the tree height (see algorithm `AddLeaf`, step 6), so the tree $T_1^{(i-1)}$ resulted from the splicing has height $ht(s_{i-1}) + 1$, again not larger than $ht(s_i)$. This concludes that we will always have $ht(T_1^{(i-1)}) \leq ht(s_i)$, so the lemma is proved. □

Now we are ready for the following theorem

Theorem 2.3.4 *The algorithm* `Split` *runs in time* $O(\log n)$.

PROOF. It is obvious that steps 1, 2, 3, and 5 of the algorithm `Split` take time $O(\log n)$. Thus, to prove the theorem, we only need to prove that the `While` loops in steps 4 and 6 of the algorithm take time $O(\log n)$.

We first consider, for each i, the amount of time spent on splicing the 2-3 tree $T_1^{(i-1)}$ and the subtrees in the segment s_i to get the 2-3 tree $T_1^{(i)}$. By Lemma 2.3.3, $ht(T_1^{(i-1)}) \leq ht(s_i)$. If s_i is a single subtree τ_i, then by Theorem 2.3.2, the time for splicing $T_1^{(i-1)}$ and τ_i to get $T_1^{(i)}$ is bounded by a constant times $ht(s_i) - ht(T_1^{(i-1)})$.

Now suppose that s_i consists of two subtrees τ_i' and τ_i'', and that the tree $T_1^{(i-1)}$ is first spliced with τ_i' that gives a tree τ_i^+, then the tree τ_i^+ is spliced with τ_i'' to get $T_1^{(i)}$. The time for splicing $T_1^{(i-1)}$ and τ_i' to get τ_i^+ is again bounded by a constant times $ht(s_i) - ht(T_1^{(i-1)})$. Moreover, the height of the resulting tree τ_i^+ is either $h(s_i)$ or $h(s_i)+1$. So splicing τ_i^+ with τ_i'' of height $ht(s_i)$ takes only constant time. Therefore, in this case, the total time to construct $T_1^{(i)}$ from $T_1^{(i-1)}$ and s_i is bounded by a constant times $ht(s_i) - ht(T_1^{(i-1)}) + 1$.

In summary, to construct the 2-3 tree $T_1 = T_1^{(q)}$, the time of the `While` loop in step 4 of the algorithm `Split` (noticing that the tree $T_1^{(1)}$ can always be constructed from s_1 in constant time) is bounded by a constant times

$$\sum_{i=2}^{q} (ht(s_i) - ht(T_1^{(i-1)}) + 1)$$

By Lemma 2.3.3, $ht(s_{i-1}) \leq ht(T_1^{(i-1)})$ for all i. Thus, the time complexity of the `While` loop in step 4 is bounded by a constant times

$$\sum_{i=2}^{q} (ht(s_i) - ht(s_{i-1}) + 1) = ht(s_q) - ht(s_1) + (q-1)$$

Since the quantities $h(s_q)$, $h(s_1)$, q are all bounded by $\log n$, we conclude that the `While` loop in step 4 takes time $O(\log n)$. The same conclusion applies for step 6 of the algorithm, thus completing the proof of the theorem. □

2.4 Geometric graphs in the plane

A graph $G = (V, E)$ is *planar* if it can be embedded in the plane without edge crossings. A *planar embedding* of a planar graph $G = (V, E)$ is a mapping of each vertex in V to a point in the plane and each edge in E to a simple curve between the images of the two endpoints of the edge, so that no two images of edges intersect except at their endpoints. The image of the mapping is

called a *geometric graph* in the plane. If a geometric graph G is connected, then it determines a subdivision of the plane, in which each region is also called a *face*. There is an unbounded region in the subdivision that contains the infinite point of the plane.

If all edges of a geometric graph G are straight-line segments in the plane, G is called a *planar straight-line graph* (PSLG). For a connected PSLG, each bounded region together with the edges of G that are on the boundary of the region, forms a polygon in the plane. Our study is in general focused on PSLGs. In most cases, we assume that the PSLGs are connected.

2.4.1 Euler's formula

Let n, e and f denote the number of vertices, the number of edges, and the number of regions (including the unbounded region) of a connected PSLG G, respectively. The famous Euler's formula relates these parameters by

$$n - e + f = 2.$$

if we have an additional property that each vertex has degree at least 3 then we can prove the following relations.

$$\begin{aligned} n &\leq 2e/3 && (2.3)\\ f &\leq 2e/3 && (2.4)\\ e &\leq 3f - 6 && (2.5)\\ e &\leq 3n - 6 && (2.6)\\ n &\leq 2f - 4 && (2.7)\\ f &\leq 2n - 4 && (2.8) \end{aligned}$$

To see relation (2.3), observe that each vertex has degree at least 3, and each edge contributes exactly 2 to vertex degrees, one for each of its endpoints. Thus, the sum of vertex degrees is equal to $2e$ and is at least as large as $3n$, which gives (2.3). Similarly, relation (2.4) is derived from the observation that each edge has two *edge sides* that are used on region boundaries, while each region uses at least three edge sides on its boundary. To see relation (2.5), we start with Euler's formula $e = n + f - 2$, and apply inequality (2.3), which gives $e/3 \leq f - 2$ thus (2.5). The relation (2.6) is obtained similarly using Euler's formula and inequality (2.4). Finally, relation (2.7) comes from (2.3) and (2.5), and relation (2.8) comes from (2.4) and (2.6).

Thus, for planar graphs, the number of vertices, the number of edges, and the number of regions are linearly related. This is very different from general graphs, where a graph of n vertices may have up to $n(n-1)/2$ edges.

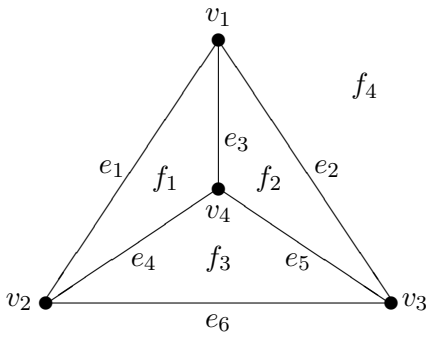

Figure 2.1: A planar embedding of K_4

2.4.2 Doubly connected edge list (DCEL)

Consider a planar embedding I of the complete graph K_4, as depicted in Figure 2.1. What information should we keep for this embedding? Of course, the vertices and edges of K_4 should be recorded. Moreover, it is necessary to keep the information about the regions of the embedding I. To represent the information of a region, we must know what is the edge sequence when we traverse the boundary of the region. For example, suppose that in Figure 2.1 we are traversing the boundary of region f_3 from vertex v_2 to vertex v_4 along edge e_4 (thus, the region is on the right side when we traverse). When we reach vertex v_4, we must know what is the next edge on the boundary of f_3 (here this should be edge e_5). From the figure, we can see that the next edge e_5 must be the next edge obtained by rotating the current traversed edge e_4 counterclockwise around the vertex v_4. Thus, to get the region boundaries, we must know the cyclic ordering for the edges incident on each vertex of the embedding I.

The *doubly connected edge list (DCEL)* is an efficient data structure for representing a PSLG. The main component of DCEL for a PSLG G is the *edge nodes*. There is a one-to-one correspondence between the edges of G and edge nodes in the corresponding DCEL. An edge node consists of four information fields V_1, V_2, F_1 and F_2, and two pointer fields P_1 and P_2. The fields V_1 and V_2 contain the starting vertex and ending vertex of the edge, respectively (so we give each edge of the PSLG G an orientation, which can be defined arbitrarily). The fields F_1 and F_2 contain the names of the regions which lie, respectively, to the left and right of the edge oriented from V_1 to V_2. The pointer P_1 (resp. P_2) points to the edge node for the first edge encountered after the edge (V_1, V_2) when one proceeds counterclockwise around V_1 (resp. V_2). Therefore, the edge P_1 is the edge following the edge

(V_1, V_2) at the vertex V_1, while the edge P_2 is the edge following the edge (V_1, V_2) at the vertex V_2 in the embedding of G.

The following is the DCEL for the PSLG given in Figure 2.1.

	V_1	V_2	F_1	F_2	P_1	P_2
e_1	v_2	v_1	f_4	f_1	e_6	e_3
e_2	v_1	v_3	f_4	f_2	e_1	e_5
e_3	v_1	v_4	f_2	f_1	e_2	e_4
e_4	v_4	v_2	f_3	f_1	e_5	e_1
e_5	v_3	v_4	f_3	f_2	e_6	e_3
e_6	v_2	v_3	f_3	f_4	e_4	e_2

Note that the space used by a DCEL to represent a PSLG is linear to the number of edges of the PSLG.

Suppose that the vertices of a PSLG G are v_1, ..., v_n, and that the regions of G are f_1, ..., f_m. We use two additional arrays $HV[1..n]$ and $HF[1..m]$ for the DCEL of G, where for each $i = 1, \cdots, n$, $HV[i]$ points to an edge node in the DCEL such that one edge end of the corresponding edge is v_i, and for $h = 1, \cdots, m$, $HF[h]$ points to an edge node in the DCEL such that the corresponding edge is on the boundary of the region f_h, .

Using the DCEL of a PSLG G, we can efficiently traverse the boundary edges of a region of G (in the order in which the edges appear in the traversing) or the edges incident on a vertex of G (in the cyclic order of the edges around the vertex). The following is an algorithm for traversing the boundary of a region when the DCEL of G is given. The algorithm for traversing the edges incident on a vertex of G can be given similarly.

```
Algorithm  Trace-Region(i)
/* Trace the boundary edges of the region i. */
1. e = HF[i];
2. e' = e;
3. If (DCEL[e][F1]==i)
     e = DCEL[e][P1];
   Else e = DCEL[e][P2];
4. While (e <> e') Do
     If (DCEL[e][F1]==i)
       e = DCEL[e][P1]
     Else e = DCEL[e][P2];
```

For example, if we start with `HF[3]` $= 4$, and use the DCEL given above for the planar imbedding of the complete graph K_4 in Figure 2.1, we will get the boundary for region f_3 as $[e_4, e_5, e_6]$.

It is easy to see that the algorithm `Trace-Region` spends constant time on each edge when traversing the region boundary of a region. Therefore,

the time complexity of the algorithm `Trace-Region` is $O(h)$, where h is the *size* of the region, i.e., the number of edges on the boundary of the region.

Note that if the rotation of the edges incident on each vertex of the PSLG G is given in counterclockwise order in the DCEL for G, then the regions are traversed clockwise by the algorithm `Trace-Region` (i.e., the region is on the right side during the traversing). On the other hand, if the rotation of the edges incident on each vertex of the PSLG G is given in clockwise order in the DCEL for G, then the regions are traversed counterclockwise by the algorithm `Trace-Region` (i.e., the region is on the left side during the traversing). Moreover, it is easy to see that for a PSLG G, a DCEL for G in which the rotation of the edges incident on each vertex of G is given in counterclockwise order can be transformed in linear time into a DCEL for G in which the rotation of the edges incident on each vertex of G is given in clockwise order, and vice versa. The detailed implementation of this transformation is left to the reader as an exercise.