Advanced Algorithms and Datastructures - Full Notes

André O. Andersen

2021

Max Flow

General Knowledge

Proofs

Examples

Linear Programming and Optimization

General Knowledge

Proofs

Examples

Hashing

General Knowledge

Proofs

Examples

Van Emde Boas Trees

General Knowledge

Proofs

Examples

NP-Completeness

General Knowledge

Proofs

Examples

Exact Exponential Algorithms and Parameterized Complexity

General Knowledge

Proofs

Examples

Approximation Algorithms

General Knowledge

We have at least three waays to get around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, we may be able to isolate important special cases that we can solve in polynomial time. Third, we might come up with approaches to find *near-optimal* solutions in polynomial time. We call an algorithm that returns near-optimal solutions an ***approximation algorithm***

We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n). \tag{1}$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a ***$\rho(n)$-approximation algorithm***. The approximation ratio of an approximation algorithm is never less than 1, since $\frac{C}{C^*} \leq 1$ implies $\frac{C^*}{C} \geq 1$. Therefore, a 1-approximation algorithm produces an optimal solution.

An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not opnly an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1+\epsilon)$-approximation algorithm. We say that an approximation scheme is a ***polynomial-time approximation scheme*** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

We say that an approximation scheme is a ***fully polynomial-time approximation scheme*** if it is an approximation scheme and its running time is polynomial in both $\frac{1}{\epsilon}$ and the size n of the input instance. For example, the scheme might have a running time of $O\left(\left(\frac{1}{\epsilon}\right)^2 n^3\right)$. With such a scheme, any decrease in ϵ comes with an instace in the running time.

We say that a randomized algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the expected cost C of the solution procuded by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

We call a randomized algorithm that achieves an approximation ratio of $\rho(n)$ a ***randomized $\rho(n)$-approximation algorithm***

Examples

The Vertex-cover Problem

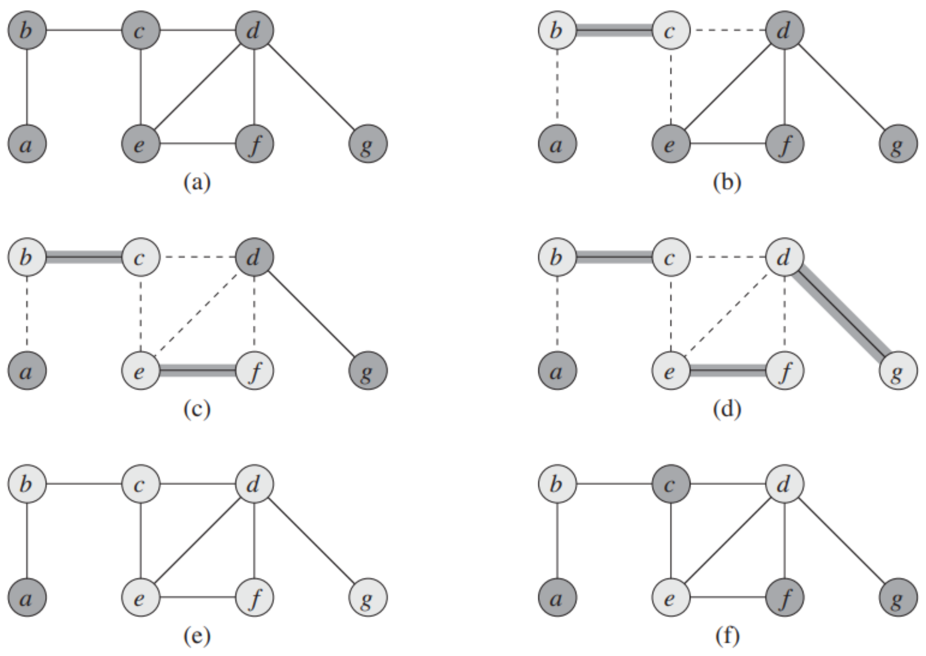

Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen; vertices e and f are added to C. **(d)** Edge (d, g) is chosen; vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is a nedge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it. The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***.

The approximation algorithm `APPROX-VERTEX-COVER`(G) returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover. The running time of this algorithm is $O(V + E)$, using adjacency lists to represent E'

The traveling-salesman problem

In the traveling-salesman problem, we are given a complete undirected graph $G = (V, E)$ that has a non-negative integer cost $c(u, v)$ associated wit heach edge $(u, v) \in E$, and we must find a hamiltonian cycle of G with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$.

We say that the cost function c satisfies the ***triangle inequality*** if, for all vertices $u, v, w \in V$

$$c(u, w) \leq c(u, v) + c(v, w).$$

Algorithm 1 APPROX-VERTEX-COVER

```
Require: Undirected graph $G$
1: $C = \emptyset$
2: $E' = G.E$
3: while $E' \neq \emptyset$ do
4:     let $(u, v)$ be an arbitrary edge of $E'$
5:     $C = C \cup \{u, v\}$
6:     remove from $E'$ edge $(u, v)$ and every edge incident on either $u$ or $v$
7: return $C$
```

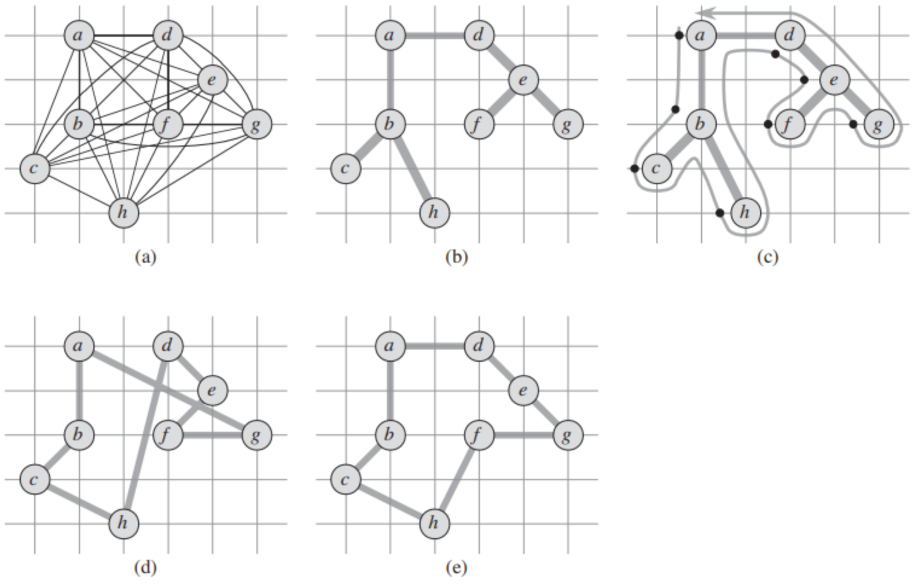

Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour H^* for the original complete graph. Its total cost is approximately 14.715.

We shall first compute a minimum spanning tree, whose weight gives a lower bound on the le ngth of an optimal traveling-salesman tour. We shall then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality. `APPROX-TSP-TOUR`(G, c) implements this approach with a running time of $\Theta(V^2)$ depending on how a simple implementation of how to compute the minimum spanning tree

Algorithm 2 APPROX-TSP-TOUR

```
Require: Complete undirected graph
Require: Cost function $c$ satisfying the triangle inequality
1: select a vertex $r \in G.V$ to be a "root" vertex
2: compute a minimum spanning tree $T$ for $G$ from root $r$
3: let $H$ be a list of vertices, ordering according to when they are first visited in a preorder tree walk of $T$
4: return the hamiltionian cycle $H$
```

The set-covering problem

An instance (X, F) of the ***set-covering problem*** consists of a finite set X and a family F of subsets of X, such that every element of X belongs to at least one subset in F. We say that a subet $S \in F$ ***covers*** its elements. The problem is to find a minimum-size subset $C \subseteq F$ whose members cover all of X (both F and C are thus sets of multiple sets). We say, that any C that cover all of X, ***covers*** X.

The greedy method works by picking, at each stage, the set S that covers the greatest number of remaining elements that are uncovered We can easily implement the algorithm to run in time polynomial in $|X|$ and $|F|$.

Algorithm 3 GREEDY-SET-COVER

```
Require: A finite set $X$
Require: A family $F$ of subsets of $X$
1: $U = X$
2: $C = \emptyset$
3: while $U \neq \emptyset$ do
4:     Select an $S \in F$ that maximizes $|S \cap U|$
5:     $U = U - S$
6:     $C = C \cup \{S\}$
7: return $C$
```

Since the number of iterations of the loop is bounded from above by $\min(|X|, |F|)$, and we can implement the loop body to run in time $O(|X||F|)$, a simple implementation runs in time $O(|X||F| \min(|X|, |F|))$.

MAX-3-CNF satisfiability

A particular instance of 3-CNF satisfiability may or may not be satisfiable. In order to be satisfiable, there must exist an assignment of the variables so that every clause evaluates to 1. If an instance is not satisfiable, we may want to compute how "close" to satisfiable it is, that is, we may wish to find an assignment of the variables that satisfies as many clauses as possible. We call the resulting maximization problem ***MAX-3-CNF satisfiability***. The input to MAX-3-CNF satisfiability is the same as for 3-CNF satisfiability, and the goal is to return an assignment of the variables that maximizes the number of clauses evaluating to 1. We require each clause to consist of exactly three distinct literals. We further assume that no clause contains both a variable and its negation.

$$\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} w(v)\, x(v) \\
\text{subject to} \quad & \\
& x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \\
& x(v) \leq 1 \quad \text{for each } v \in V \\
& x(v) \geq 0 \quad \text{for each } v \in V\,.
\end{aligned}$$

Vertex cover using linear programming

In the ***minimum-weight vertex-cover problem***, we are given an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. for any vertex cover $V' \subseteq V$, we define the weight of the vertex cover $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight. We shall compute a lower bound on the weight of the minimum-weight vertex cover, by using a linear program. We shall then "round" this solution and use it to obtain a vertex cover.

Suppose, that we associate a variable $x(v)$ with each vertex $v \in V$, and let us require that $0 \leq x(v) \leq 1$ for each $v \in V$. We put v into the vertex cover iff $x(v) \geq 1/2$. Then, we can write the constraint that for any edge (u, v), at least one of u and v must be in the vertex cover as $x(u) + x(v) \geq 1$. This view gives rise to the ***linear-programming relaxation*** for finding a minimum-weight vertex cover The procedure `APPROX-MIN-WEIGHT-VC`(G, w) uses the solution to the linear-programming relaxation to construct an approximate solution to the minimum-weight vertex-cover problem

Algorithm 4 APPROX-MIN-WEIGHT-VC

```
Require: Undirected graph $G = (V, E)$
Require: Positive weight function $w(v)$ for each $v \in V$
1: $C = \emptyset$
2: Compute $\bar{x}$, an optimal solution to the lienar program
3: for each $v \in V$ do
4:     if $\bar{x}(v) \geq 1/2$ then $C = C \cup \{v\}$
5: return C
```

Proofs

Theorem 35.1

`APPROX-VERTEX-COVER` is a polynomial-time 2-approximation algorithm

Proof It has been shown in a previous chapter, that it runs in polynomial time.
The set C of vertices that is returned by the algorithm is a vertex cover, since the laogrithm loops until every edge in $G.E$ has been covered by some vertex in C.
Let A denote the set of edges that line 4 picked. Not two edges in A share an endpoint. Thus no two edges in A are covered by the same vertex from an optimal cover C^*, and we have the lower bound

$$|C^*| \geq |A| \tag{2}$$

on the size of an optimal vertex cover. Since A consists of the edges between two vertices in C (and since all of the elements in C are unique), we have the (exact) upper bound on the size of the vertex cover returned

$$|C| = 2|A| \tag{3}$$

Combining equation (2) and (3), we obtain

$$|C| = 2|A| \leq 2|C^*|$$

Theorem 35.2

`APPROX-TSP-TOUR` is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality

Proof It has been shown in a previous chapter, that it runs in polynomial time.
Let H^* denote an optimal tour for the given set of vertices. We obtain a spanning tree by deleting any edge fro ma tour, and each edge cost is nonnegative. Therefore, the weight of the minimum spanning tree T provides a lower bound on the cost of an optimal tour

$$c(T) \leq c(H^*). \tag{4}$$

A ***full walk*** of T lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let us call this full walk W. Since the full walk traverses every edge of T exactly twice, we have

$$c(W) = 2c(T) \tag{5}$$

Inequality (4) and equation (5) imply that

$$c(W) \leq 2c(H^*) \tag{6}$$

and so the cost of W is within a factor of 2 of the cost of an optimal tour.
Unfortunately, the full walk W is generally not a tour, since it visits some vertices more than once. By the triangle inequality, however, we can delete a visit to any vertex from W and the cost does not increase (If we delete a vertex v from W between visits to u and w, the resulting ordering specifies going directly from u to w). By repeatedly applying this operation, we can remove from W all but the first visit to each vertex. This ordering is the same as that obtained by a preorder walk of the tree T. Let H be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since every vertex is visited exacly once, and in fact it is the cycle computed by the algorithm. Since H is obtained by deleting vertices from the full walk W, we have

$$c(H) \leq c(W). \tag{7}$$

Combining inequalities (6) and (7) gives $c(H) \leq 2c(H^*)$, which completes the proof.

Theorem 35.4

`GREEDY-SET-COVER` is a polynomial-time $\rho(n)$-approximation algorithm, where

$$\rho(n) = H(\max\{|S| : S \in F\})$$

and $H(d) = \sum_{i=1}^{d} 1/i$ is the dth harmonic number.

Proof It has been shown in a previous chapter, that it runs in polynomial time.
To show that the algorithm is a $\rho(n)$-approximation algorithm, we assign a cost of 1 to each set selected by the algorithm, distribute this cost over the elements covered for the first time, and then use these costs to derive the desired relationship between the size of an optimal set cover C^* and the size of the set cover C returned by the algorithm. Let S_i denote the ith subset selected by the algorithm. The algorithm incurs a cost of 1 when it adds S_i to C. We spread this cost of selection S_i evenly among the elements covered for the first time by S_i. Let c_x denote the cost allocated to element x, for each $x \in X$. Each element is assigned a cost only once, when it is covered for the first time. If x is covered for the first time by S_i, then

$$c_x = \frac{1}{|S_i - (S_i \cup S_2 \cup ... \cup S_{i-1}|}.$$

Each step of the algorithm assigns 1 unit of cost, and so

$$|C| = \sum_{x \in X} c_x. \tag{8}$$

Each element $x \in X$ is in at least one set in the optimal cover C^*, and so we have $\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x$. Combining equation (**??**) and inequalitiy (**??**), we have that

$$|C| \leq \sum_{s \in C^*} \sum_{x \in S} c_x. \tag{9}$$

The remainder of the proof rests on the following key inequality, which we shall prove shortly. For any set S belonging to the family F

$$\sum_{x \in S} c_x \leq H(|S|). \tag{10}$$

From inequalities (**??**) and (**??**) it follows that

$$|C| \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \cdot H(\max\{|S| : S \in F\}),$$

thus proving the theorem.
All that remains is to pro ve inequality (10). Consider any set $S \in F$ and any $i = 1, 2, ..., |C|$, and let $u_i = |S - (S_1 \cup S_2 \cup ... \cup S_i)$ be the number of elements in S that remain uncovered after the algorithm has selected sets $S_1, S_2, ..., S_i$. We define $u_0 = |S|$ to be the number of elements of S, which are all initially uncovered. Let K be the least index such that $u_k = 0$, so that every element in S is uncovered by at least one of the sets $S_1, S_2, ..., S_k$ and some element in S is uncovered by $S_1 \cup S_2 \cup ... \cup S_{k-1}$. Then, $u_{i-1} \geq u_i$, and $u_{i-1} - u_i$ elements of S are covered for the first time by S_i, for $i = 1, 2, ..., k$. Thus,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_i \cup S_2 \cup ... \cup S_{i-1})|}.$$

Observe that

$$|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1}) \geq |S - (S_1 \cup S_2 \cup ... \cup S_{i-1}| = u_{i-1},$$

because the greedy choice of S_i guarantees that S cannot cover more new elements than S_i does. Consequently, we obtain

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

We now bound this quantitiy as follows

$$\sum_{x\in S} c_x \le \sum_{i=1}^{k}(u_{i-1}-u_i)\frac{1}{u_{i-1}}$$

$$=\sum_{i=1}^{k}\sum_{j=u_i+1}^{u_i-1}\frac{1}{u_{u_i-1}}$$

$$\le\sum_{i=1}^{k}\sum_{j=u_i+1}^{u_i-1}\frac{1}{j} \quad \text{(because } j \le u_{u_i-1} \text{)}$$

$$=\sum_{i=1}^{k}(H(u_{i-1})-H(u_i))$$

$$=H(u_0)-H(u_k) \quad \text{(because the sum telescopes)}$$

$$=H(u_0)-H(0)$$

$$=H(u_0) \quad \text{(because } H(0)=0\text{)}$$

$$=H(|S|)$$

which completes the proof of inequalities (10).

Corollary 35.5

`GREEDY-SET-COVER` is a polynomial-time $(\ln|X|+1)$-approximation algorithm.

Proof Use inequality (A.14) and Theorem Theorem 35.4

Theorem 35.6

Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, ..., x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability 1/2 and to 0 with probability 1/2 is a randomized 8/7-approximation algorithm.

proof Suppose that we have independently set each variable to 1 with probability 1/2 and to 0 with probability 1/2. For $i = 1, 2, ..., m$, we define the indicator random variable

$$Y_i = I\{\text{clause } i \text{ is satisfied}\}$$

so that $Y_i = 1$ as long as we have set at least one of the literals in the ith clause to 1. Since no literal appears more than once in the same clause, and since we have assumed that no variable and its negation appear in the smae clause, the settings of the three literals in each clause are independent. A clause is not satisfied only if all three of its literals are set to 0, and so $Pr\{\text{clause } i \text{ is not satisfied}\} = (1/2)^3 = 1/8$. Thus, we have $Pr\{\text{clause } i \text{ is satisfied}\} = 1 - 1/8 = 7/8$, and $E[Y_i] = 7/8$. Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + ... + Y_m$. Then, we have

$$E[Y] = E\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} E[Y_i] = \sum_{i=1}^{m} 7/8 = 7m/8$$

. Clearly, m is an upper bound on the number of satisfied clauses, and thence the approximation ratio is at most $m/(7m/8) = 8/7$.

Theorem 35.7

Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

Proof Because there is a polynomial-time algorithm to solve the lienar program in line 2, and because the for-loop runs in polynomial time, the algorithm is a polynomial-time algorithm.
Nowe we show that the algorithm is a 2-approximation algorithm. Let C^* be an optimal solution to the minimum-weight vertex-cover problem, and let z^* be the value of an optimal solution to the linear program. Since an optimal vertex cover is a feasible solution to thel inear program, z^* must be a lower bound on $w(C^*)$, that is,

$$z^* \leq w(C^*). \tag{11}$$

Next, we claim that by rounding the fractional values of the variables $\bar{x}(v)$, we produce a set C that is a vertex cover and satisfies $w(C) \leq 2z^*$. To see that C is a vertex cover, consider any edge $(, v) \in E$. By the first constraint, we know that $x(u) + x(v) \geq 1$, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2$. Therefore, at least one of u and v is included in the vertex cover, and so every edgge is covered.
Now, we consider the weight of the cover. We have

$$z^* = \sum_{v \in V} w(v)\bar{x}(v) \geq \sum_{v \in V : \bar{x} \geq 1/2} w(v)\bar{x}(v) \geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} = \sum_{v \in C} w(v) \cdot \frac{1}{2} = \frac{1}{2} \sum_{v \in C} w(v) = \frac{1}{2} w(C)$$

$$= \frac{1}{2} w(C). \tag{12}$$

Combining inequalities (11) and (12) gives

$$w(C) \leq 2z^* \leq 2w(C^*)$$

and hence APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm.

Polygon Triangulation

General Knowledge

This topic is about the *Art Gallary problem*; how many caomeras do we need to guard a given gallary and how do we decide to place them? We model a gallary as a polygonal region in the plane. We further restrict ourselves to geions that are *simple polygons* (that is, a polygon of a single chain that does not intersect itself).

Let $\mathcal{P}$ be a simple polygon with n vertices. Because $\mathcal{P}$ may be a complicated shape, it seems difficult to say anything about the number of cameras we need to guard $\mathcal{P}$. Hence, we first decompose $\mathcal{P}$ into pieces that are easy to guard, namely triangles. We do this by drawing diagonals between pair of vertices.

A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals is called a *triangulation* of the polygon. We require that the set of non-intersecting diagonals to be maximal to ensure that no triangle has a polygon vertex in the interior of one of its edges. We can thus guard $\mathcal{P}$ by placing a camera in every triangle of a triangulation $T_{\mathcal{P}}$ of $\mathcal{P}$.

Placing cameras at vertices seems better, because a vertex can be incident to many triangles, and a camera at that vertex guards all of them. This suggest the following approach: let $T_{\mathcal{P}}$ be a triangulation of $\mathcal{P}$. Select a subset of the vertices of $\mathcal{P}$, such that any triangle in $T_{\mathcal{P}}$ has at least one selected vertex, and place the cameras at the selected vertices.

To find such a subset we assign each vertex of $\mathcal{P}$ a color: white, grey or black. The coloring will be such that any vertices connected by an edge or a diagonal have different colors. this is called a *3-coloring* of a triangulated polygon. In a 3-coloring of a triangulated polygon, every triangle has a white, black and gray vertex. Hence, if we place cameras at all gray vertics, say, we have guarded the whole polygon. By choosing the smallest color class to place the cameras, we guard $\mathcal{P}$ using at most $\lfloor \frac{n}{3} \rfloor$ cameras.

Let P be a simple polygon with n vertices. A set of $\lfloor \frac{n}{3} \rfloor$ camera position in The 3-coloring approach is optimal in worst case such that any point inside The 3-coloring approach is optimal in worst case is visible from at least one of the cameras can be computed in $O(n \log n)$ time.
The previous triangulation algorithm will take quadratic time in worst case. This can be improved for some classes of polygons. For instance, convex polygons: Pick one vertex and draw diagonals from every other vertex, that is nto a neighbor, to the vertex. This runs in linear time. Thus, a possible solution would be to decompose The 3-coloring approach is optimal in worst case into convext pieces and then triangulate the pieces. However, it is difficult to partition a polygon into convex pieces. Therefore, we shall decompose The 3-coloring approach is optimal in worst case into *monotone pieces*, which is a lot easier.

A simple polygon is called *monotone with respect to a line* L if for any line L' perpendicular to L, the intersection of the polygon with L' is connected. A polygon that is monotone with respect to the y-axis is called *y-monotone.* The following property is characteristic for y-monotone polygons: if we walk from a topmost to a bottommost vertex along the left (or the right) boundary chain, then we always move downwards or horizontally, never upwards.

Our strategy to triangulate the polygon $\mathcal{P}$ is to first partition $\mathcal{P}$ into y-monotone pieces, and then triangulate the pieces. We can partition a polygon into monotone pieces as follows. imagine walking from the topmost vertex of $\mathcal{P}$ to the bottommost vertex on either boundary chain. A vertex where the direction in which we walk switches from downward to upward or from upward to downward is called a *turn vertex.* To partition $\mathcal{P}$ into y-monotone pieces, we need to remove the turn vertices, which is done by adding diagonals. If at a turn vertex v both incident edges go down and the interior of the polygon lies above v, then we must choose a diagonal that goes up from v. v cannot be a turn vertex in either of the two resulting subpolygons. If both incident edges of a turn vertex go up and the interior lies bellow it, we have to choose a diagonal that goes down.

We distinguish five types of vertex in P. Four of these types are turn vertices: *Start vertices*, *Split vertices*, *End vertices*, and *merge vertices*. They are defined as follows. A vertex is a start vertex if its two neighbors lie below it and the interior angle at v is less than π. If the interior angle is greater than π then v is a split vertex. A vertex is an end vertex if its two neighbors lie above it and the interior angle at v is less than π. If the interior angle is greater than π then v is a merge vertex. The vertices that are not turn vertices are *regular vertices.* Thus a regular vertex has one of its neighbors above it and the other neighbor below it.

Lemma 3.4 implies, that $\mathcal{P}$ has been partitioned into y-monotone pieces once we get rid of its split and merge vertices. We do this by adding a diagonal going upward from each split vertex and downward from each merge vertex. Once done, $\mathcal{P}$ has been split into y-monotone pieces.

Let's see how we can add the diagonals for the split vertices. Let $v_1, v_2, ..., v_n$ be a counterclock enumeration of the vertces of $\mathcal{P}$. Let $e_1, ..., e_n$ be the set of edges of $\mathcal{P}$, where $e_i = \overline{v_i v_{i+1}}$ for $1 \leq i < n$ and $e_n = \overline{v_n v_1}$.

The algorithm moves a line L downward over the plane. The line halts at certain event points; in our case the vertices of $\mathcal{P}$. The event points are stored in an event queue Q. The event queue is a priority queue, where the priority of a vertex is its y-coordinate. This way the next event to be handled can be found in $O(\log n)$ time. The goal of the sweep is to add diagonals from each split vertex to a vertex lying above it. Let e_j be the dge immediately to the left of v_i on the sweep line, and let e_k be the edge immediately to the right of v_i on the sweep line. Then we can always conenct v_i to the lowest vertex in between e_j and e_k, and above v_i. If there is no such vertex then we can connect v_i to the upper endpoint of e_j or to the upper endpoint of $_k$. This vertex is called the *helper* of e_j and is denoted by $helper(e_j)$. Formally, $helper(ej)$ is defined as the lowest vertex above the sweepline such that the horizontal segment connecting the vertex to e_j lies inside $\mathcal{P}$

Nowe we know how to get rid of split vertices. What about merge vertices. Suppose the sweep line reaches a merge vertex v_i. let e_j and e_k be the edge simmediately to the right and left of v_i on the sweepline. Observe that v_i becomes the new helper of e_j when we reach it. We would like to connect v_i to the highest vertex below the sweep line in between e_j and e_k. We do not know the highest vertex below the sweep line when we reach v_i. But it is easy to find later on: when we reacj a vertex v_m, that replaces v_i as the hlper of e_j, then this is the vertex we are looking for. So whenever we replace the hlper of some edge, we check whether the old helper is amerge vertex and, if so, we add the diagonal between the old helper and the new one. This diagonal is always added when the new helper is a split vertex, to get rid of the split vertex. It can also happen that the hlper of e_j is not replaced anymore below v_i. In this case we can connect v_i to the lower endpoint of e_j.

In the above approach, we need to ifnd the dge to the left of each vertex. Therefore we store the edges of $\mathcal{P}$ intersecting the sweep line in the leaves of a dynamic binary search tree T. The left-to-right order of the leaves of T corresponds to the left-to-right order of the edges. Because we are only interested in edges to the left of split and merge vertices we only need to store edges in T that have the interior of P to the right. With each edge in T store its helper. The tree T and the helpers stored with the edges form the status of the sweep line algorithms.

The algorithm partitions $\mathcal{P}$ into subpolygons that have to be processed in a later stage. To have easy access to these subpolygons we shall store the subdivision induced by $\mathcal{P}$ and the added diagonals in a doubly-connected edge-list D. We assume P is initially specified as a doubly-connected edge-list. The diagonals outputed for the split and merge vertices are added to the doubly-connected edge-list. To access the doubly-connected edge-list we use cross-pointer between the edges in the status structure and the corresponding edges in the doubly-connected edge-list. Adding a diagonal can then be done in constant time.

Running time: Constructing Q takes linear time and initializing T takes constant time. To handle an event during the sweep we perform one operation on Q, at most one query, one insertion, and one deletion on T, and we insert at most two diagonals into D. Priority queues and balanced search trees allow for queries and updates in $O(\log n)$ time, and an insert into D takes $O(1)$ time. Hence, handling and event takes $O(\log n)$ time, and the algorithm runs in $O(n \log n)$ time.

In the following we show, that monotone polygons can be triangulated in linear time. Together with the partition into monotone pieces in $O(n \log n)$ time, this implies that any simple polygon can be triangulated in $O(n \log n)$ time.

Let $\mathcal{P}$ be an y-monotone polygon with n vertices. We assume that $\mathcal{P}$ is *strictly y-monotone*, that is, it does not contain horizontal edges.

This property is what makes triangulation a monotone polygon easy: we can work our way through $\mathcal{P}$ from top to the bottom on both chains, adding diagonals whenever this is possible.

The triangulation algorithm handles the vertices in order of decreasing y-coordinate. The algorithm requires a stack S as auxilary data structure. Initially the stack is empty; later it contains the vertices of P that have been encountered but may still need more diagonals. When we handle a vertex we add as many diagonals from this vertex to vertices on the stack as possible. These diagonals split off triangles from $\mathcal{P}$. The vertices that have been handled but not split off - the vertices on the stack - are on the boundary of the part of $\mathcal{P}$ that still needs to be triangulated. The lower the vertex (that is, the more recently it has been encountered), the higher on the stack it is. The part of $\mathcal{P}$ that still needs to be triangulated has a particular shape: it looks like an upside down funnel. One boundary of the funnel consists of a part of a single edge of $\mathcal{P}$, and the other boundary consists of reflex vertices (interior angle is atleast 180^o). Only the highest vertex (which is at the bottom of the stack) is convex. This property remains true after, we have handled the next vertex, hence, it is an invariant of the algorithm.

Ket's see which diagonals we can add when we handle the next vertex. We distinguish two cases: v_j, the next vertex to handle, lies on the same chain as the reflex vertices on the stack, or it lies on the opposite chain

1. If v_j lies on the opposite chain, it must be the funnel. We can add diagonals from v_j to all vertices correctly on the stack, except for the last one; this is the upper vertex, so it is already connected to v_j. All the vertices are popped from the stack v_j and the vertex previously on top of the stack are pushed onto the stack
2. The other case is when v_j is on the same chain as the reflex vertices on the stack. First, pop one vertex from the stack; this vertex is already connected to v_j by an edge of $\mathcal{P}$. Next, pop vertex from the stack and connect them to v_j untill we encounter one where this is not possible. When we find a vertex where we cannot connect v_j, we push the last vertex that has been popped back onto the stack. After this has been done we push v_j onto the stack

What is the running time of this algorithm? Linear - see pseudocode.

Proofs

Theorem 3.1

Every simple polygon admits a triangulation and any triangulation of a simple polygon with n vertices consists of exactly $n-2$ triangles.

Proof
We prove this theorem by inducting on n. When $n=3$ the polygon itself is a triangle and the theorem is trivially true. Let $n>3$ and assume that the theorem is true for all $m<n$. Let $\mathcal{P}$ be a polygon with n vertices

We first prove the existence of a diagonal in $\mathcal{P}$. Let v be the leftmost vertex of $\mathcal{P}$. Let u and w be the two neighboring vertices of v on the boundary of $\mathcal{P}$. If the open segment $\overline{uw}$ lies in the interior of $\mathcal{P}$, we have found a diagonal. Otherwise, there are one or more vertices inside the triangle defined by u, v and w, or on the diagonal $\overline{uw}$. Of those vertices, let v' be the one furthest from the line through u and w. The segment connecting v' and v cannot intersect an edge of $\mathcal{P}$, because such an edge would have an endpoint inside the triangle that is farther from the line through u and w, contradicting the definition of v'. Hence, $\overline{vv'}$ is a diagonal.

Any diagonal cuts $\mathcal{P}$ into two simple subpolygons $\mathcal{P}_1$ and $\mathcal{P}_2$. Let m_1 be the number of vertices in $\mathcal{P}_1$ and m_2 be the number of vertices in $\mathcal{P}_2$. both m_1 and m_2 must be smaller than n, so by induction $\mathcal{P}_1$ and $\mathcal{P}_2$ can be triangulated. Hence, $\mathcal{P}$ can be triangulated as well.

It remains to prove that any triangulation of $\mathcal{P}$ consists of $n-2$ triangles. Consider an arbitrary diagonal in some triangulation $T_{\mathcal{P}}$. This diagonal cuts $\mathcal{P}$ into two subpolygons with m_1 and m_2 vertices, respectively. Every vertex of $\mathcal{P}$ occurs in exactly one of the two subpolygons, except the vertices defining the diagonal, which occurs in both. Hence, $m_1 + m_2 = n + 2$. By induction, any triangulation of $\mathcal{P}_i$ consists of $m_i - 2$ triangles, which implie sthat $\mathcal{P}$ consists of $(m_1 - 2) + (m_2 - 2) = n - 2$ triangles.

Theorem 3.2

The 3-coloring approach is optimal in worst case.

Proof A 3-coloring always exists. To see this, we look at the *dual graph* of $T_{\mathcal{P}}$. This graph $g(T_{\mathcal{P}})$ has a node for every triangle $T_{\mathcal{P}}$. We denote the triangle corresponding to a node v by $t(v)$. There is an arc between two nodes v and μ if $t(v)$ and $t(\mu)$ share a diagonal. the arcs in $g(T_{\mathcal{P}})$ correspond to diagonals in $T_{\mathcal{P}}$. Because any diagonal cuts $\mathcal{P}$ into two, the removal of an edge from $g(T_{\mathcal{P}})$ splits the graph into two. Hence, $g(T_{\mathcal{P}})$ is a tree. This means that we can find a 3-coloring using a simple graph traversal, such as depth first search.

While we do the depth first search, we maintain the following invariant

1. All vertices of the already encountered trinagles have been colored white, black or gray
2. No two connected vertices have received the same color.

This implies that we have computed a valid 3-coloring when all triangles have been encountered.

The depth first search can be started from any node of $g(T_{\mathcal{P}})$; the three vertices of the corresponding triangle are colored white, gray and black. Now suppose, we reach a node v in g, coming from a node μ. Hence $t(v)$ and $t(\mu)$ share a diagonal. Since the vertices of $t(\mu)$ have already been colored, only one vertex remains to be colored. Because $g(T_{\mathcal{P}})$ is a tree, the other nodes adjacent to v have not been visited yet, and we still have the freedom to give the vertex the remaining color.

We conclude that the 3-coloring approach is optimal in worst case.

Lemma 3.4

A polygon is y-monotone if its has no split vertices or merge vertices.

Proof Suppose $\mathcal{P}$ is not y-monotone. We have to prove that $\mathcal{P}$ contains a split or a merge vertex. Since $\mathcal{P}$ is not monotone, there is a horizontal line $\mathcal{P}$ that intersects $\mathcal{P}$ in more than one connected component. We can choose L such that the leftmost component is a segment and not a single point. Let p be the left endpoint of this segment, and let q be the right endpoint.

Starting at q, we follow the boundary of $\mathcal{P}$ such that p lies to the left of the boundary (this means that we go up from q). At some point r, we will go up again. If $r \neq p$, then the highest vertex we encountered while going from q to r must be a split vertex. If $r = p$, we again follow the boundary of $\mathcal{P}$ starting at q, but in the other direction.

Let r' be the point where the boundary intersects L. We cannot have $r' = p$, because that would mean that the boundary of $\mathcal{P}$ intersects L only twice, contradicting that L intersects $\mathcal{P}$ in more than one component. So we have $r' \neq p$, implying that the lowest vertex we encountered going from q to r' must be a merge vertex.

Lemma 3.5

The algorithm for transforming a polygon monotone adds a set of non-intersecting diagonals that partitions $\mathcal{P}$ into monotone subpolygons.

Proof NOT DONE.