CS 170 HW 14

Due on 2019-05-06, at 9:59 pm

1 Study Group

List the names and SIDs of the members in your study group.

2 Convex Hull

Given n points in the plane, the *convex hull* is the list of points, in counter-clockwise order, that describe the convex shape that contains all the other points. Imagine a rubber band is stretched around all of the points: the set of points it touches is the convex hull.

In this problem we'll show that the convex hull problem and sorting reduce to each other in linear time.

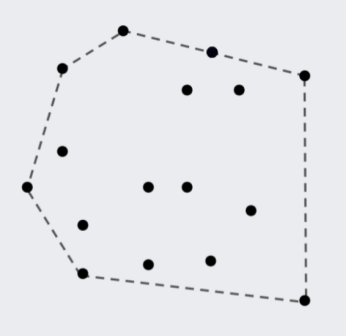


Figure 1: An instance of convex hull: the convex hull is the seven points connected by dashed lines. Note that three or more points in the convex hull can be collinear.

(a) Fill in the following algorithm for convex hull; you do not need to prove it correct. What is its runtime? For simplicity, in this part you may assume no three points are collinear.

procedure ConvexHull(list of points $P[1..n]$)

Set $low :=$ the point with the minimum y-coordinate, breaking ties by minimum x-coordinate.

Create a list $S[1..n-1]$ of the remaining points sorted increasingly by the angle between the vector $point - low$ and the vector $(1, 0)$ (i.e the x-axis) .

Initialize $Hull := [low, S[1]]$

for $p \in S[2..n-1]$ **do**

<fill in the body of the loop>

Return $Hull$

(b) Now, find a linear time reduction from sorting to convex hull. In other words, given a list of real numbers to sort, describe an algorithm that transforms the list of numbers into a list of points, feeds them into convex hull, and interprets the output to return the sorted list. Then, prove that your reduction is correct.

Solution:

(a) **procedure** ConvexHull(list of points $P[1..n]$)

Set low := the point with the minimum y-coordinate, breaking ties by minimum x-coordinate.

Create a list $S[1..n-1]$ of the remaining points sorted increasingly by the angle between the vector $point - low$ and the vector $(1, 0)$ (i.e the x-axis) .

Initialize $Hull := [low, S[1]]$

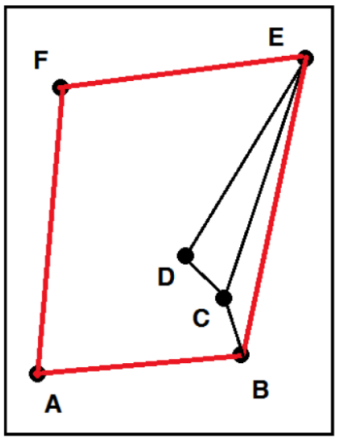
for $p \in S[2..n-1]$ **do**

While the angle between $Hull[-2]$, $Hull[-1]$, and p is a right turn*, pop $Hull[-1]$.

Append p to the end of $Hull$.

Return $Hull$

* The angle between a, b, and c is a right turn if $\overrightarrow{ab}$ is counterclockwise from $\overrightarrow{bc}$.



For example, in this diagram, $\overrightarrow{ABC}$ and $\overrightarrow{BCD}$ are left turns, but $\overrightarrow{CDE}$ is a right turn, so we pop D. Then, $\overrightarrow{BCE}$ is a right turn, so we pop C. $\overrightarrow{ABE}$ and $\overrightarrow{BEF}$ are left turns, so the complete convex hull, shown in red, is $ABEF$.

It takes $\Theta(n)$ time to compute the slope of the points from low, on which we can sort. Then, it takes $\Theta(n \log n)$ time to sort the points. The loop appends each point to the hull exactly once, and pops it from the hull at most once, and makes a constant number of operations before each push or pop. So the loop takes $\Theta(n)$ time in total. Therefore, the reduction to sorting (all the nonsorting steps) take $\Theta(n)$ time, and the algorithm as a whole takes $\Theta(n \log n)$ time.

(b) Transform the list of numbers into points on the plane as follows: $f(n) = (n, 0)$. Then add the dummy node (0, -1) to this list. Run convex hull on the list. Note that it can return the hull starting with any point, as long as the points are in counterclockwise order. Therefore, shift the list so that (0,-1) is first (while the first point isn't (0,-1), move the first point to the end of the list). Then, remove (0,-1) from the result, transform the points in the result back into numbers by removing the y-coordinate, reverse their order, and finally return the new list.

This reduction is correct because we can be certain that we transform the points into an instance of convex hull with a solution that is exactly the sorted list of points. No matter the set of points (as long as it's at least two points), the construction will lead to a triangle, so all the points will be on the convex hull (no points will be inside the hull). After we shift the points so that (0,-1) is first, the other points will be in reverse order of x-coordinate, because they must be in counterclockwise order. Therefore, reversing the order of the result will yield the sorted list.

In our reduction, transforming numbers to points (and back) takes constant time, and reversing the list takes linear time. Thus the reduction takes linear time in total.

3 Decision vs. Search vs. Optimization

The following are three formulations of the VERTEX COVER problem:

- As a *decision problem*: Given a graph G, return TRUE if it has a vertex cover of size at most b, and FALSE otherwise.
- As a *search problem*: Given a graph G, find a vertex cover of size at most b (that is, return the actual vertices), or report that none exists.
- As an *optimization problem*: Given a graph G, find a minimum vertex cover.

At first glance, it may seem that search should be harder than decision, and that optimization should be even harder. We will show that if any of the above variants of Vertex-Cover can be solved in polynomial time, all the other variants are also solvable in polynomial time.

For the following parts, describe your algorithms precisely; justify correctness and the number of times that the black box is queried (asymptotically).

(a) Suppose you are handed a black box that solves VERTEX COVER (DECISION) in polynomial time. Give an algorithm that solves VERTEX COVER (SEARCH) in polynomial time.

(b) Similarly, suppose we know how to solve VERTEX COVER (SEARCH) in polynomial time. Give an algorithm that solves VERTEX COVER (OPTIMIZATION) in polynomial time.

Solution:

(a) If given a graph G and budget b, we first run the DECISION algorithm on instance (G, b). If it returns "FALSE", then report "no solution".

If it comes up "TRUE", then there is a solution and we find it as follows:

- Pick any node $v \in G$ and remove it, along with any incident edges.
- Run DECISION on the instance $(G \setminus \{v\}, b-1)$; if it says "TRUE", add v to the vertex cover. Otherwise, put v and its edges back into G.
- Repeat until G is empty.

Correctness: If there is no solution, obviously we report as such. If there is, then our algorithm tests individual nodes to see if they are in any vertex cover of size b (there may be multiple). If and only if it is, the subgraph $G \setminus \{v\}$ must have a vertex cover no larger than $b-1$. Apply this argument inductively.

Running time: We may test each vertex once before finding a v that is part of the b-vertex cover and recursing. Thus we call the DECISION procedure $O(n^2)$ times. This can be tightened to $O(n)$ by not considering any vertex twice. Since a call to DECISION costs polynomial time, we have polynomial complexity overall.

Note: this reduction can be thought of as a greedy algorithm, in which we discover (or eliminate) one vertex at a time.

(b) Binary search on the size, b, of the vertex cover.

Correctness: This algorithm is correct for the same reason as binary search.

Running time: The minimum vertex cover is certainly of size at least 1 (for a nonempty graph) and at most $|V|$, so the SEARCH black box will be called $O(\log |V|)$ times, giving polynomial complexity overall.

Finally, since solving the optimization problem allows us to answer the decision problem (think about why), we see that all three reduce to one another!

Note that the reductions here are slightly different from what we will typically use because we are allowed to query the oracle (black box) multiple times here. As a result we have not actually shown the optimization problem to be in **NP**. Instead, we can only say that it is in a (possibly) larger complexity class known as $\mathbf{P}^{\mathbf{NP}}$. This is slightly beyond the scope of this course.

4 2-SAT and Variants

In this problem we will explore the variant of 2-SAT called Max-2-SAT. Recall that 2-SAT is in instance of SAT where each clause contains at most 2 literals (hereby called a 2-clause). As seen previously, 2-SAT can be solved efficiently via a graph theoretic techniques and is hence in P. We will show that a subtle modification of the original 2-SAT problem renders the problem computationally hard.

The problem of Max-2-SAT is defined as follows. Let $C_1, \ldots, C_m$ be a collection of 2-clauses and k a non-negative integer. We want to determine if there is some assignment which satisfies at least k clauses.

The problem of Max-Cut is defined as follows. Let G be an undirected unweighted graph, and k a non-negative integer. We want to determine if there is some cut with at least k edges crossing it. Max-Cut is known to be NP-complete.

Show that Max-2-SAT is NP-complete by reducing from Max-Cut. Prove the correctness of your reduction.

Solution:

As suggested, we reduce unweighted `Max-Cut` to `Max-2-SAT`. Let a graph $G = (V, E)$ and an integer k be given. We want to find whether there is a cut in the graph of size k or more.

We construct a boolean formula based on the graph. Every assignment of this formula will correspond to a cut in the graph.

For each vertex $u \in V$, we add a variable x_u, representing whether u is in S (true) or in $V \setminus S$ (false). For each edge $(u, v) \in E$, we add *two* clauses: $(x_u \vee x_v)$ and $(\overline{x_u} \vee \overline{x_v})$. Then, if (u, v) crosses the cut, *both* of these clauses will be satisfied (if $u \in S$ and $v \notin S$, then x_u and $\overline{x_v}$ are both true; if $u \notin S$ and $v \in S$, then $\overline{x_u}$ and x_v are both true). If (u, v) does not cross the cut, then exactly one of the clauses will be satisfied (the first if both u and v are in S, the second if neither is). Thus, any cut of size q in the graph corresponds to an assignment of values to variables that satisfies exactly $|E| + q$ clauses (1 for each of the $|E| - q$ pairs representing non-cut edges, 2 for each of the q pairs representing cut edges).

Thus, to solve $\texttt{Max-Cut}(G, k)$, we construct this formula Φ_G and return $\texttt{Max-2-SAT}(\Phi_G, |E| + k)$.

Lastly, we know that `Max-2-SAT` is in NP because given an instance and an assignment of variables, we can just iterate through the clauses and count how many are satisfied in polynomial time.