CS 170 HW 11

Due **2019-11-14, at 9:59 pm**

1 Study Group

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

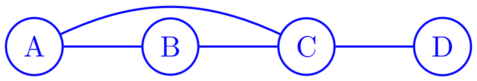
2 A Reduction Warm-up

In the Rudrata path problem (aka the Hamiltonian Path Problem), we are given a graph G and want to find if there is a path in G that uses each vertex exactly once.

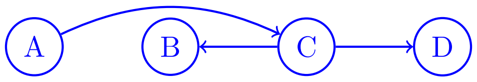
Is the following argument correct? Please justify your answer.

We will show that Undirected Rudrata Path can be reduced to Longest Path in a DAG. Given a graph G, use DFS to find a traversal of G and assign directions to all the edges in G based on this traversal (i.e. edges will point in the same direction they were traversed and back edges will be omitted). This gives a DAG. If the longest path in this DAG has $|V|-1$ edges then there is a Rudrata path in G since any simple path with $|V|-1$ edges must visit every vertex.

Solution: It is incorrect.
It is true that if the longest path in the DAG has length $|V|-1$ then there is a Rudrata path in G. However, to prove a reduction correct, **you have to prove both directions**. That is, if you have reduced problem A to problem B by transforming instance I to instance I' then you should prove that I has a solution **if and only if** I' has a solution. In the above "reduction," one direction doesn't hold. Specifically, if G has a Rudrata path then the DAG that we produce does not necessarily have a path of length $|V|-1$–it depends on how we choose directions for the edges.
For a concrete counterexample, consider the following graph:



It is possible that when traversing this graph by DFS, node C will be encountered before node B and thus the DAG produced will be



which does not have a path of length 3 even though the original graph did have a Rudrata path.

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 one can be solved in polynomial time, so can the others.
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.

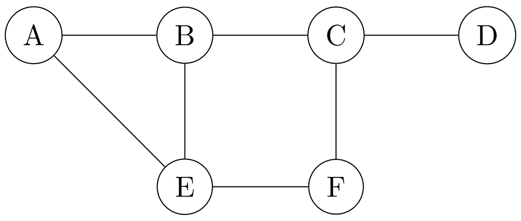
Correctness: This algorithm is correct for the same reason as binary search.

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 Reducing Vertex Cover to Set Cover

In the minimum vertex cover problem, we are given an undirected graph $G = (V, E)$ and asked to find the smallest set $U \subseteq V$ that "covers" the set of edges E. In other words, we want to find the smallest set U such that for each $(u, v) \in E$, either u or v is in U (U is not necessarily unique). For example, in the following graph, $\{A, E, C, D\}$ is a vertex cover, but not a minimum vertex cover. The minimum vertex covers are $\{B, E, C\}$ and $\{A, E, C\}$.



Recall the following definition of the minimum Set Cover problem: Given a set U of elements and a collection $S_1, \ldots, S_m$ of subsets of U, what is the smallest collection of these sets whose union equals U? So, for example, given $U := \{a, b, c, d\}$, $S_1 := \{a, b, c\}$, $S_2 := \{b, c\}$, and $S_3 := \{c, d\}$, a solution to the problem is the collection of S_1 and S_3.
Give an efficient reduction from the Minimum Vertex Cover Problem to the Minimum Set Cover Problem.

Solution: Let $G = (V, E)$ be an instance of the Minimum Vertex Cover Problem. Create an instance of the Minimum Set Cover Problem where $U = E$ and for each $u \in V$, the set S_u contains all edges adjacent to u. Let $C = \{S_{u_1}, S_{u_2}, \ldots, S_{u_k}\}$ be a set cover. Then our corresponding vertex cover will be $u_1, u_2, \ldots, u_k$. To see this is a vertex cover, take any $(u, v) \in E$. Since $(u, v) \in U$, there is some set S_{u_i} containing (u, v), so u_i equals u or v and (u, v) is covered in the vertex cover.
Now take any vertex cover $u_1, \ldots, u_k$. To see that $S_{u_1}, \ldots, S_{u_k}$ is a set cover, take any $(u, v) \in E$. By the definition of vertex cover, there is an i such that either $u = u_i$ or $v = u_i$. So $(u, v) \in S_{u_i}$, so $S_{u_1}, \ldots, S_{u_k}$ is a set cover.
Since every vertex cover has a corresponding set cover (and vice-versa) and minimizing set cover minimizes the corresponding vertex cover, the reduction holds.

5 Convex Hull

Given n points in the plane, the *convex hull* is the list of points, in counter-clockwise order, that describe the convex polygon 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. You can also play around with defining your own set of points and seeing what the polygon should look like at `http://cs.yazd.ac.ir/cgalg/AlgsVis/ConvexHull.html`
In this problem we'll show that the convex hull problem and sorting reduce to each other in linear time.

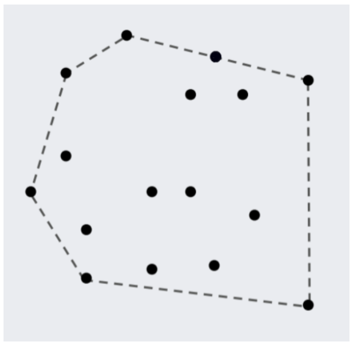


Figure 1: An instance of convex hull: the convex hull is the seven points connected by dashed lines. Note that three or more points in the convex hull can be collinear.

(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.

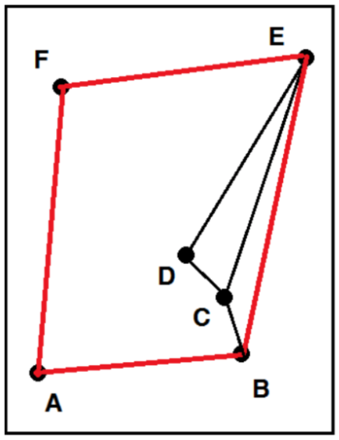
```
procedure ConvexHull(list of points $P[1..n]$)
    Set $low$ := the point with the minimum $y$-coordinate, breaking ties by minimum $x$-coordinate.
    Create a list $S[1..n-1]$ of the remaining points sorted increasingly by the angle between the vector $point - low$ and the vector $(1, 0)$ (i.e 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.