ALGORITHMS I NOTES

BRENDAN WHITAKER

2010 *Mathematics Subject Classification.* 12-XX

Abstract. A comprehensive set of notes for Professor Lai's Algorithms I course, taken SP18 at The Ohio State University.

Contents

Part 1

CHAPTER 1

Mathematical Foundations

Tuesday, January 9th

The following is correct:

$$9n^2 + 6n + 9 = O(n^2),$$

But the equality in this case is ordered, it is meaningless to write:

$$O(n^2) = 9n^2 + \cdots,$$

so don't!. Don't ever write the $O(n^2)$ on the left.

THEOREM 1. *If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then:*
(1) $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$
(2) $f_1(n) + f_2(n) \in O(max(g_1(n), g_2(n))$
(3) $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$.

There is a similar theorem for Ω and Θ, and both use max.

REMARK 1.1. Take note of the **approximating summations by integrals** slide, and the formulas on it. It is new material.

Solve the running time for binary search with a recurrence relation. You should know how to work with his notation.

Tuesday, January 16th

We do exercises from 1.2 Example Analysis.

EXERCISE 5.

The inner for loop executes $2n^3/i$. So we have:

$$T(n) = \sum_{i=1}^{3} n^2 = 2n^3 \left(\sum_{i=1}^{3} n^2 \right) = \Theta(n^3 log_n).$$

But consider if we make the inner for loop go to $2n^3/i$. Then doing the above analysis again, we would get $nlogn$, but this is wrong. You need the running time of the inner for loop to always be at least 1, because it is always at least 1. So we write:

$$\sum_{i=1}^{3} n^2 \left(2n/i + 1\right) = \cdots = \Theta(n^2).$$

Now we discuss divide and conquer. You should remember that quicksort and mergesort are divide and conquer algorithms.

Be careful of floor and ceilings. Professor Lai wants us to assume that n is some power, and then apply Theorem 7 to then generalize to all n. Note that the function must be asymptotically non-decreasing for us to apply this theorem.

You should know your $o(f(n))$ and $\omega(g(n))$ definitions. Are they stronger or weaker than their capital counterparts? They have something to do with limits. If the limit goes to 0 or ∞.

In order of strength: $O(f(n)), o(f(n)), << f(n)$, where this last one denotes "polynomially smaller".

When using Master Theorem, you can just ignore the floor or ceiling functions.

You can basically always assume the weird condition for case 2 is satisfied without checking it.

Wednesday, January 18th

When recurrences involve roots, it suffices to only consider powers of the parameter of the root.

Given:

$$T(n) = \begin{cases} 2T(\sqrt{n}) + logn & n > 2 \\ c & otherwise \end{cases}$$

So we define $S(m) = T(2^m) = T(n)$. We then get:

$$S(m) = \begin{cases} 2S(m/2) + m & m > 1 \\ c & otherwise \end{cases}$$

then by master theorem, we get $S(m) \in \Theta(mlog(m))$, so $T(n) \in \Theta(lognloglogn)$. (Why?)

REMARK 1.2. If you're trying to find the running time of a recursive algorithm and you have no idea how to do it, use a recursion tree.

1.1. 2331 HW2

The inner while loop executes $\frac{i^2-6}{\sqrt{i}}$ times and thus takes $ci^{3/2}$ time. Let $k = 7^{3/2}$.

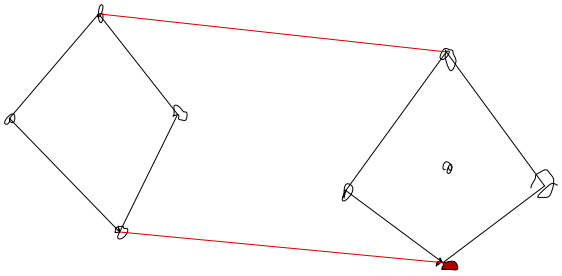
$$\begin{aligned} T(n) &= 25c(1 + k + k^2 + k^3 + \cdots + k^{log_7(n^3-25)}) \\ &= 25c\frac{1 - k^{log_7(n^3-25)+1}}{1-k} \\ &= \frac{25c}{k-1}\left(k^{log_7(n^3-25)+1} - 1\right) \\ &\leq \frac{25ck}{k-1}\left(k^{log_7(n^3-25)}\right) \\ &\leq c'k^{3log_7 n} \\ &= c'(k^{log_7 n})^3 \\ &= c'((7^{3/2})^{log_7 n})^3 \\ &= c'n^{9/2} \in O(n^{9/2}) \\ T(n) &\geq 25c(7^{3/2})^{log_7(n^3-25)} = 25c(7^{log_7(n^3-25)})^{3/2} \\ &= 25c(n^3-25)^{3/2} \in \Omega(n^{9/2}) \\ &\approx (k^{log_7 n})^3 = (7^{log_7 n})^{9/2} = n^{9/2} \in \Theta(n^{9/2}). \end{aligned} \tag{1.1}$$

Thursday, January 25th

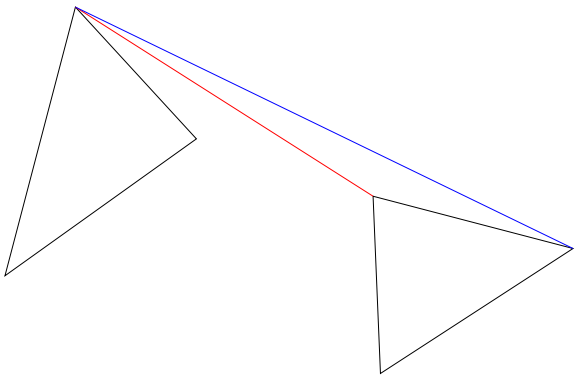
We discuss the convex hall problem.

DEFINITION 1.3. The convex hull is the smallest convex polygon that encloses all points of A. Our straightforward algorithm is for each line segment betweeen two points in A, check if every point is on one side of the line. Checking each line segment takes n^2 time, and checking each point for each line segment takes n time, so we have a running time of n^3.

But we can do this by divide and conquer in $O(nlogn)$.
We do this by dividing it along a vertical line into two sets if we have more than 3 points. Then we find the convex hull of the two subsets and then connect them by the upper and lower bridges (red lines). How do we find these? Note that it is note necessarily the points with the highest y coordinates.

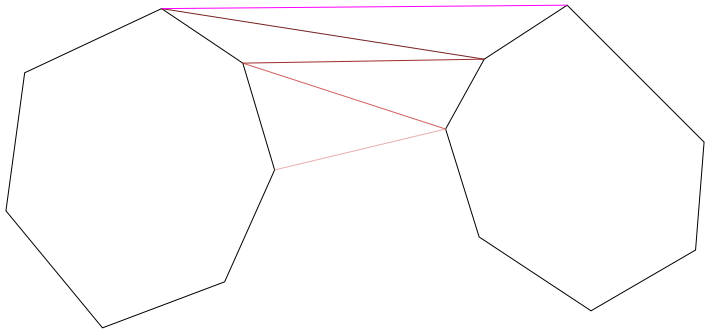


We give a counterexample:



Note that the red line is between the two points with highest y coordinate, but the blue line is the actual upper bridge.

Now we describe an algorithm for finding the upper/lower bridge. We first start with the line between the points with closest x coordinates between the two sets, and then we move each point counterclockwise on the left set and clockwise on the right side until the new points no longer lie above the line between the previous:



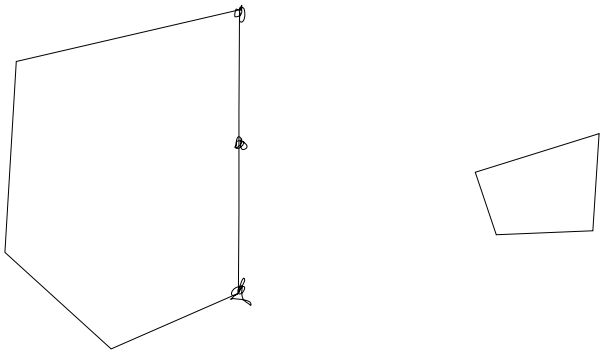
Observe that we start with the lightest pink line, and move upwards until we find the upper bridge, outline in magenta. A good question for the midterm exam is why if we randomly pick v, w why it doesn't work. Ask about this.

So what data structure do we want to use to represent the points in each set. We want to use a **linked list**. Where a point points to the next counterclockwise point and so on, and it is cyclic. So to find the leftmost and rightmost points, we could just run over all n, or we could initialize two pointers, one to the leftmost, and one to the rightmost. We found the upper bridge up $O(n)$ time, so the running time is given by:

$$T(n) = 2T(n/2) + O(n).$$

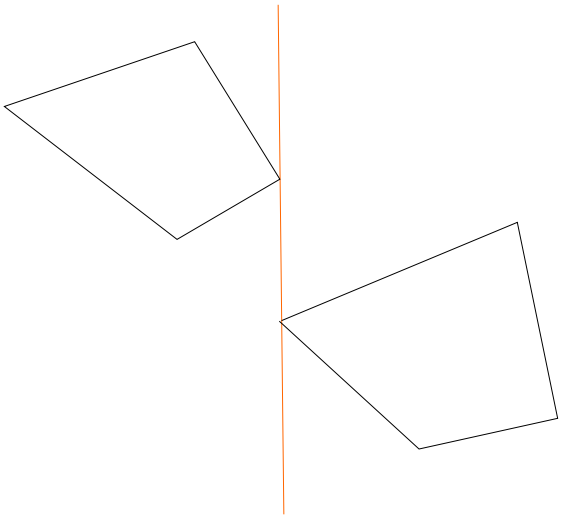
If we want to check whether a point is above or below the line, we can write the equation of the line between points u, v as $f(x, y) = ax + by + c = 0$, and compute $f(x_0, y_0)$. If it has the same sign as b, it is above the line, and if it has opposite sign, it is below.

So now we relax the assumption that no 3 points are colinear, and that they all have distinct x coordinates. So we have the situation:



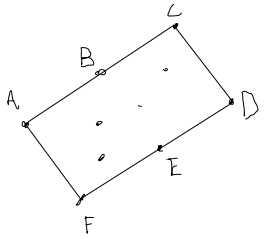
So in this case, we have ties when we are picking the rightmost point in the left set, just pick a random one. And that is how we fix that problem.

New problem, what if we have this:



So when we are looking for our upper bridge, how do we define whether or not the next counterclockwise point is "above" this line or not? It is vertical. So our solution is to put any points which all lie along the midpoint in the same set, either on the left or the right figure.

We treat the following case:



If all points in A are colinear, we call it the base case.

In finding the upper bridge, we loop over moving the points of the line (u, v), and if we choose to move if the neighbor is on or above the line, we get (A,C,D,F), but if we only move when the neighbor is above the line, we get (A,C,B,D,E,F) in the above diagram.

REMARK 1.4. Orientation of three points. (p_1, p_2, p_3) in that order is counterclockwise if:

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} > 0.$$

Clockwise if the determinant is negative, and colinear of the determinant is zero.

We discuss Homework 4, question 5. The two arrays have been sorted, we can assume distinct integers. We want to find the n-th smallest. We are looking for the element which is bigger than $n-1$ elements, and smaller than exactly n elements. This should be very similar to binary search.

CHAPTER 2

Dynamic Programming

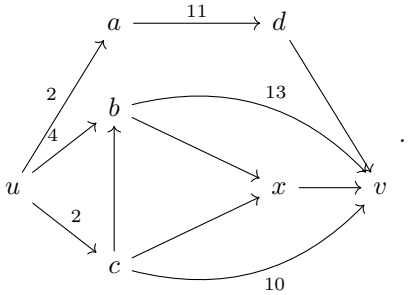
Note every problem can be solved by dynamic programming. The ones that these work for are typically optimization problems. We will usually by optimizing an objective function. The **closest pair** problem is an optimization problem. The **convex hull** problem is also one.

DEFINITION 2.1. **Optimization problem:** Construct a set/sequence of elements $\{ y_1, ..., y_k \}$ that satisfies a given constraint and optimizes an objective function.

REMARK 2.2. The closest pair problem takes a set $\{ p_1, ..., p_n \}$ of points and finds a subset $\{ p_i, p_j \}$ such that $p_i \neq p_j$ (constraint) and minimizes the distance between them (objective function).

REMARK 2.3. The convex hull problem takes a set of points $\{ y_1, ..., y_n \}$ and finds a subset $\{ y_i, ..., y_l \}$ such that this set forms a convex polygon that contains all points. Our objective function $f(y_i, ..., y_l)$ is the area of the polygon, and we are trying to minimize it.

Dynamic Programming for Shortest Path Problem.
Consider the directed acyclic graph (DAG):



.

So for x_1, we have j options, in this case nodes a, b, c. Each of these options lead to a subproblem. So define $P(u, v)$ as the problem of finding a shortest path from $u \to v$. Then if we choose a as x_1 our problem becomes $P(a, v)$ and the second parameter will always be v, so we can represent each problem as $P(s)$ where s is the current node, and the destination node of v is implicit. Now we want to define the objective function to be optimized using these parameters. Let $f(x)$ denote the distance from node $x \to v$. So we have the problem $P(s)$ and the function $f(s)$. Now we formulate a recurrence relation:

$$f(x) = \min(\{ d(x, y) + f(y) : (x, y) \in E \}), \tag{2.1}$$

if $x \neq v$ and out-degree$(x) \neq 0$. So we can also define:

$$f(x) = \begin{cases} 0 & \text{if } x = v \\ \infty & \text{if } x \neq v \text{ and out-degree}(x) = 0 \end{cases}. \tag{2.2}$$

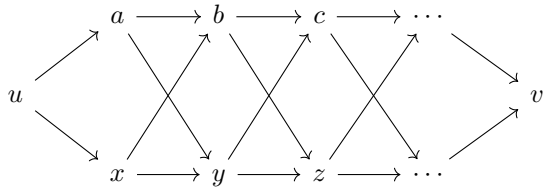
Our goal is to compute $f(u)$ then it will be easy to construct the shortest path. **Compute the objective function first!**

So we immediately have the following algorithm for computing $f(u)$:

Algorithm 1: shortest(x)

```
1  begin
2      global d[1..n, 1..n];
3      if x = v then
4          return 0;
5      else if out-degree(x) = 0 then
6          return ∞;
7      else
8          return min{ d(x, y) + shortest(y) : (x, y) ∈ E };
9      end
10 end
```

Consider the DAG:



So note that our function is called once at u, once at a, x and twice at each of b, y and 4 times at each of c, z and so on. So Since there are about $\frac{n}{2}$ levels, we know that the worst case running time is actually exponential, since we have $1, 2, 4, 8, 16, ...$ and other shit. So we revise our algorithm:

Algorithm 2: shortest(x)

```
1  begin
2      global d[1..n, 1..n], F[1..n], Next[1..n];
3      if F[x] = −1 then
4          if x = v then
5              F[x] ⟵ 0;
6          else if out-degree(x) = 0 then
7              F[x] ⟵ ∞;
8          else
9              F[x] ⟵ min{ d(x, y) + shortest(y) : (x, y) ∈ E };
10             Next[x] ⟵the node y that yielded the min;
11         end
12     end
13     return F[x];
14 end
```

Thursday, February 1st

Dynamic programming overview. We have $x_1, x_2, ..., x_k$. And we have several options $a, b, c...$ for x_1. And we choose the best option out of these, and we describe the original problem as a function using minimal parameters.

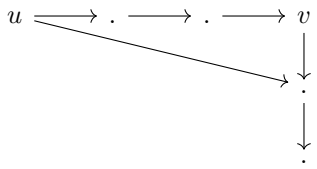
Steps for dynamic programming on exam.

(1) Define $f()$.
(2) Write a recurrence:
$$f(x) = min\{ ... \}.$$
(3) Boundary conditions. This is "like" the base case, when you know the value of f immediately.
(4) goal: Compute $f(u)$.
(5) Algorithms (implement).
    Algorithm 1: write a procedure to actually compute $f(x), \forall x$ (shortest distance).

Algorithm 2: print the shortest path.

(6) Time complexity.

How many times is shortest called. Consider:



Note we might want to say the number of edges reachable from u, but if they are after v in a path, we won't get to them. So then we might want to say number of edges reachable from u but not from v, but the path in the above diagram is reachable from both and we still call it. So we should write the number of edges reachable from u such that we do not pass v.

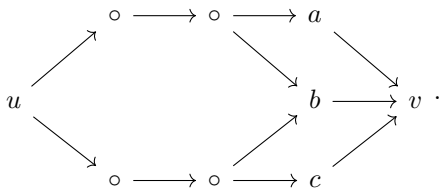
So how long does it take? The running time for $shortest(x)$ is broken into two cases:

$$T(n) = \begin{cases} c + \text{ time to find outgoing edges of x} & \text{first call} \\ c & \text{subsequent calls} \end{cases}$$

So the overall worst case running time is given by $O(|E|) \cdot O(1)+$ the time to find all nodes' outgoing edges.

To find this, with adjacency matrix this takes $O(|V|^2|)$ time. But if we use an adjacency list we need only $O(|V| + |E|)$. We really only need the number of edges time $|E|$, but what if there are no edges in the graph? Then we still need to iterate over the whole adjacency list, so we need $|V| + |E|$.

Consider $u \to x_1 \to \cdots x_k \to v$. What is x_k in general? In the following diagram it is one of a, b, c:



So this is different, now f is defined from the back, we are starting with the nodes next to v, but before we were choosing nodes next to u. There are TWO DIFFERENT FRIGGIN WAYS TO DO IT! Basically left to right or right to left, or forward/backward.

(1) Let $f(x)$ denote the shortest distance from $u \to x$.

(2) $f(x) = \min \{ f(y) + d(y, x) : (y, x) \in E \}$.

$$f(x) = \begin{cases} 0 & \text{if } x = u \\ \infty & \text{if } x \neq v \text{ and in-degree}(x) = 0 \end{cases}. \tag{2.3}$$

We have this distinction for any dynamic programming problem, not just shortest path. We call them forward/backward approach. It is possible for one of them to work and one of them to not, but mostly both do. And sometimes one is easier. But it's up to discretion.

For the Matrix-chain multiplication problem we have n matrices:

$$M_1, ..., M_n.$$

Where the dimensions of M_i are $d_{i-1} \times d_i$. We use backward approach. Let x_i be the problem of the i-th matrix multiplication that we compute. So we have $n - 1$ x's. So we know x_{n-1} is the last one. And we have $n - 1$ cuts we can make in the multiplication:

$$(M_1 \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_n).$$

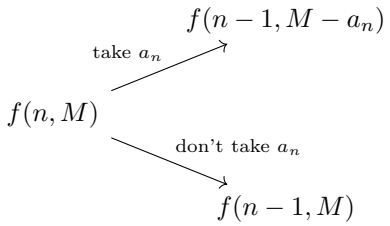
Tuesday, February 13th

Let $A = \{ 4, 6, 3, 9, 2 \}$, $M = 15$. Given a positive integer M and a multiset (allow repetition) of positive integers A, determine if there is a subset $B \subseteq A$ such that $Sum(B) = M$.

Consider:

$$a_1, a_2, ..., a_n.$$

We have:



So let $f(i, K)$ denote whether there is a subset of $\{ a_1, ..., a_n \}$ with sum equal to K.

So we have:

$$f(i,K) = \begin{cases} \max \begin{cases} f(i-1, K-a_i) & a_i \leq K \\ f(i-1,K) \end{cases} \\ f(i-1,K) & else \end{cases}$$

Boundary conditions: $f(i,0) = T$ for $i > 0$ and $f(0,K) = F$ for all K.

The running time is $O(n \cdot M)$ is **pseudo-polynomial**. Whenever the running time involves an actual value of input, then it's pseudo-polynomial.

Job scheduling on two machines. We decide that the x_i's should denote the choice of machine to run job i. Let $f(i, t_A, t_B)$ denote the min completion time for jobs $i, i+1, ..., n$ with machine A available after time t_A and same for machine B. Recurrence given by:

$$f(i, t_A, t_B) = \begin{cases} f(i+1, t_A + a_i, t_B) \\ f(i+1, t_A, t_B + b_i) \end{cases}$$

Boundary conditions: $f(n+1, t_A, t_B) = max(t_A, t_B)$. Goal: $f(1,0,0)$. The running time is pseudopolynomial because it depends on a_i, b_i.

We go over HW4 question 5. We want to find the n-th smallest element. We know the n-th smallest element is: n - 1 elements $<$ nth smallest $<$ n elements. Pretty sure I did it correctly.

We discuss the typesetting problem. We define x_i as the location of the i-th line break. Let $f(i)$ denote the minimum cost of typesetting $w_i, ..., w_n$. So:

$$f(k) = \min_{1 \leq k \leq n} \{ Cost(i,j) + f(k+1) \}.$$

CHAPTER 3

Greedy algorithms

What is a partial solution?

CHAPTER 4

Graph algorithms

If you need to determine semiconnectedness for a graph that is not acyclic, use graph of strongly connected components, and then run top sort on this, because this graph will be acyclic.

CHAPTER 5

Maximum Flow

LEMMA 5.1. *In Edmonds-Karp, for all non-source/sink nodes v, $\delta(v)$ is nondecreasing, where $\delta(v)$ is the shortest distance in number of edges from s to v.*

PROOF. Assume that it does decrease. So consider the first augmentation that decreases it. Out of all nodes for which δ decreases, let v be the node with smallest $\delta_{f'}$, where f' is the flow after this augmentation, and f is the flow before. So after the augmentation, v is closest to s out of all the nodes that had decreased δs. Then we know:

$$\delta_f(v) > \delta_{f'}(v). \tag{5.1}$$

Let p be a shortest path from s to v in the residual network $G_{f'}$ after the augmentation. Let u be the node just before v in this path. So since p is in the residual network $G_{f'}$, and (u, v) is in p, we know $(u, v) \in E_{f'}$, it's an edge in the residual network. Also, since the shortest distance from s to v is $\delta_{f'}(v)$, we know:

$$\delta_{f'}(u) + 1 = \delta_{f'}(v), \tag{5.2}$$

since the shortest distance from s to v in the new residual network certainly can't be longer than $\delta_{f'}(u) + 1$ since we have a path of this length through (u, v) which we already showed was in the residual network $G_{f'}$. And it can't be shorter than $\delta_{f'}(u) + 1$ otherwise this would violate our assumption that (u, v) is the last edge in p, the shortest path from s to v in $G_{f'}$. We also know:

$$\delta_f(u) \leq \delta_{f'}(u). \tag{5.3}$$

So we're claiming that shortest distance from s to u cannot decrease, or that u is a green node. This is because we know from (5.2) that $\delta_{f'}(u) < \delta_{f'}(v)$, so if u decreased, this would violate our construction of v, in which we said that among all nodes that decrease, v is closest to s after the augmentation.

Case 1: We assume (u, v) was in the residual network G_f before the augmentation. Then we know:

$$\delta_f(u) + 1 \geq \delta_f(v). \tag{5.4}$$

This simply says that since we have a path to u with length $\delta_f(u)$, then the shortest path to v must be shorter than the path to u plus 1, since we know $(u, v) \in G_f$. Now we have:

$$\delta_{f'}(v) = \delta_{f'}(u) + 1 \geq \delta_f(u) + 1 \geq \delta_f(v) > \delta_{f'}(v). \tag{5.5}$$

The first equality is (5.2), the second is (5.3), the third is (5.4), and the last is since distance to v is decreasing. But this is a contradiction since we are saying:

$$\delta_{f'}(v) > \delta_{f'}(v). \tag{5.6}$$

Case 2: Now we assume (u, v) is not in E_f. Since we already know $(u, v) \in E_{f'}$, we know that we must augment along a path that contains (v, u) to get from f to f'. This means that the augmenting path used to construct f' contains (v, u). Since Edmonds-Karp always augments along shortest paths, this means the shortest path from s to u contains v. This means:

$$\delta_f(v) + 1 = \delta_f(u). \tag{5.7}$$

But we have:

$$\delta_{f'}(v) + 1 < \delta_f(v) + 1 = \delta_f(u) \leq \delta_{f'}(u) = \delta_{f'}(v) - 1. \tag{5.8}$$

Which is a contradiction. □

THEOREM 5.2. *The total number of flow augmentations in Edmonds-Karp is $O(VE)$, and thus the running time is $O(VE^2)$.*

PROOF. Strategy is to show that at most $2|E|$ edges can become critical. Then show that each of these edges may become critical at most $|V|/2$ times. So we have $O(VE)$ augmentations. Then augmenting takes $O(E)$ time. So we get $O(VE^2)$.

We say an edge is **critical** on an augmenting path if its capacity is equal to the capacity of the augmenting path. Suppose an edge (u, v) becomes critical twice. Before the first augmentation we have:

$$\delta_f(u) + 1 = \delta_f(v). \tag{5.9}$$

Then later it becomes critical again. This means that it has to have positive capacity again, but it had zero capacity coming out of the first augmentation, so we needed to have augmented along (v, u) in between. Since Edmonds-Karp augments along shortest path, we know:

$$\delta_{f'}(u) = \delta_{f'}(v) + 1 \tag{5.10}$$

before the backwards augmentation. But since δ is nondecreasing by Lemma 5.1, we know:

$$\begin{aligned} \delta_{f'}(u) &= \delta_{f'}(v) + 1 \\ &\geq \delta_f(v) + 1 \quad \text{by Lemma 5.1} \\ &= \delta_f(u) + 2. \quad \text{by (5.9)} \end{aligned} \tag{5.11}$$

Thus between any two instances of (u, v) becoming critical, $\delta(u)$ increases by at least 2, meaning that since $\delta(u) \leq |V|$ we know that we have at most $|V|/2$ times each edge becomes critical.

Since we have one augmentation per time any edge becomes critical, we know we have $O(VE)$ augmentations, and since augmentations take $O(E)$ time, the running time of Edmonds-Karp is $O(VE^2)$. □

REMARK 5.3. There is a set of k edge-disjoint paths if and only if there is a flow of value k when all edges have capacity 1.

REMARK 5.4. To find the max number of node-disjoint paths, replace each node with a pair of nodes and a capacity 1 edge between them. All other edges are replaced with capacity one edges as well.

REMARK 5.5. Push is applicable when $c(u, v) > 0$ and $e(u) > 0$ and $h(u) = h(v) + 1$.

5.1. Generic Push-Relabel

LEMMA 5.6. *If $u \neq t$ is overflowing, then we can Push or Relabel.*

PROOF. If there are any edges (u, v) with positive capacity and $h(u) = h(v) + 1$, then we can push. So suppose there are no such edges. Then all edges (u, v) are such that $h(u) \leq h(v)$ which means we can relabel. □

LEMMA 5.7. *When we relabel u, its height increases by at least 1.*

PROOF. Since we are relabelling, we know:

$$h(u) \leq h(v) \tag{5.12}$$

for all edges (u, v). Let v' be the edge with minimum height. Then

$$h(u) \leq h(v'). \tag{5.13}$$

And after the relabel,

$$h(u) = h(v') + 1 \geq h(u) + 1. \tag{5.14}$$

□

LEMMA 5.8. *h is always a height function.*

PROOF. Relabel makes sure $h(u) \leq h(v) + 1$ for all outgoing edges since it chooses the minimum height v, and this is the only time the height function is modified. □

LEMMA 5.9. *f is always a preflow.*

PROOF. Note we only push the minimum of $\{ c(u, v), e(u) \}$, so the relaxed flow conversation is never violated. □

LEMMA 5.10. *If f is a preflow and we have a height function relative to f, then there is no augmenting path in G_f.*

PROOF. let p be a simple path (any path can be shortened to a simple one, and will be by the algorithm since we use BFS). Note we have:

$$\begin{aligned} h(s) - h(t) &\leq length(p) && \text{since height function decreases by at most 1 in an edge } (u,v) \\ &\leq |V| - 1. && \text{since it's simple} \end{aligned} \tag{5.15}$$

□

THEOREM 5.11. *When Generic Push-Relabel terminates, the preflow is a max flow.*

PROOF. Note f is a preflow by Lemma 5.9. And no vertex is overflowing by Lemma 5.6. So f is a flow. And h is a height function by Lemma 5.8. And there is no augmenting path in G_f by Lemma 5.10. So f is a max flow. □

5.2. Analysis of Generic

LEMMA 5.12. *Let f be a preflow, then for any overflowing vertex u, there is a path from u to s in G_f. We used up positive capacities to get to u, and we zeroed them all out on the way, so we now have capacity going back.*

Relabel-to-Front

REMARK 5.13. Preceeding each relabel, there may be at most $O(V)$ discharges.

PROOF. This is saying that we may have at most $O(V)$ discharges which do NOT relabel anything before we must have a discharge that relabels. This is because L, the linked list of all nodes, has length $O(V)$ and for each discharge we move to the next node in L, until we hit then end, and then we must either relabel to move to the front of L again, or terminate. So either way there are at most $O(V)$ discharges between relabels. Similarly, there are $O(V)$ discharges after the last relabel. □