Data Structures and Algorithms: The formal notes

Josh Felmeden

November 26, 2019

Contents

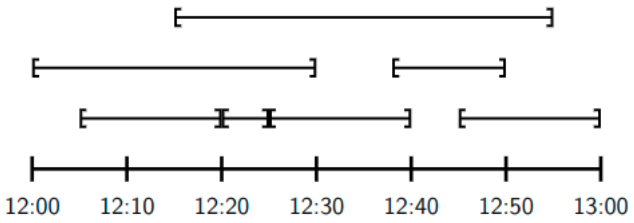
1 Greedy algorithms

1.1 Interval scheduling

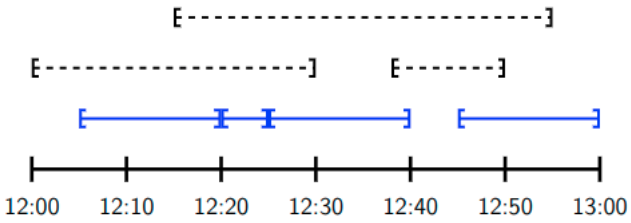
Let's suppose that you're running a satellite imaging service. Taking a satellite picture of an area isn't instant and can take some time. It can also only be done on the day where the satellite's orbit is lined up correctly. Say you want to request some images to be taken from said satellite, each of which can only be taken at certain times and you can only take one picture at a time. How do we satisfy as many requests as possible?

The requested satellite times that we have to deal with are: 12:00-12:30, 12:05-12:20, 12:15-12:55, 12:20-12:25, 12:38-12:50, and 12:45-13:00.

If we visualise this in a graph, we get:



If we take a greedy algorithm approach to assigning these slots, we could do something like assign the slot that finishes earliest, and then repeat doing this until we have reached the end. For example, the slot that finishes fastest is 12:05-12:20, so we assign this. This now removes the ability for both 12:10-12:30 and 12:15-12:55 to be assigned, so we remove these. This continues until we end up with something looking like this:



This means that we satisfy four requests (which is actually the maximum possible, so well done us).

We can formalise this by saying that a **request** is a pair of integers (s, f) with $0 \leq s \leq f$.

The algorithm that we're left with is this:

```
public sub greedySchedule
    sort R
    for each i in {1 ... n} do
        if s_i >= lastf then
            A.append(s_i, f_i)
            lastf = f_i
        end if
    next
end sub
```

Now, we need to prove that the output is actually a **compatible subset** of R. This is sort of intuitive because the set we added doesn't break compatibility, since $s \geq$ `lastf` and `lastf` is the latest finish time that's already in A.

We can formalise this with a loop invariant. At the start of the ith iteration, we see that

- A contains a compatible subset $\{(S_1, F_1), \ldots, (S_t, F_t)\}$ of R.
- `lastf` = $\max(\{0\} \cup \{F_j : j \leq t\})$

The base case ($i = 0$) is immediate because $A = []$. The induction step is that A was compatible at the start of the iteration, and therefore if we append a pair s_i, f_i to A then $s \geq$ `lastf` $\geq F_j$ for all $f \leq t$. This means that (s_i, f_i) is compatible with A.

Greedy algorithms definition

Greedy algorithms are actually an informal term and people have different definitions. The definition that we're going to use is:

- They start with a sub-optimal solution.
- They look over all the possible improvements and pick the one that looks the best at the time.
- They never backtrack in 'quality'.

Greedy algorithms might fail; it's not enough to just do the obvious thing at each stage. While the algorithms might fail initially, we can use the knowledge that we gained from the results of the algorithm to design a more correct one.

2 Graphs

Graph Definition

A **graph** is a pair $G = (V, E)$ where $V = V(G)$ is a set of **vertices** $E = E(G)$ is a set of **edges** contained in $\{\{u, v\} : u, v \in V, u \neq\}$

Walk Definition

A **walk** in a graph $G = (V, E)$ is a sequence of vertices such that $\{v_i, v_{i+1} \in E$ for all $i \leq k - 1$
We say that the walk is from v_0 to v_k and call k the length of the walk.

2.1 Euler walks

An **Euler walk** is one that contains every edge in G exactly once.

Two graphs might be **equal**. This is the case when two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are equal and written $G_1 = G_2$ if $V_1 = V_2$ and $E_1 = E_2$. This does present some issues, however, because sometimes graphs look like they should be equal, when they're not because the edges are labelled differently.

This is where **isomorphism** comes in. G_1 and G_2 are **isomorphic** if there's a bijection $f : V_1 \to V_2$ such that $\{f(u), f(v)\} \in E_2$ if and only if $\{u, v\} \in E_1$.

Intuitively, this means that $G_1 \xrightarrow{\sim} G_2$ if they are the same graph but the vertices are relabelled.

In a graph $G = (V, E)$, the **neighbourhood** of a vertex v is the set of vertices joined to v by an edge. Formally, $N_G(v) = \{w \in V : \{v, w\} \in E$. Also, for all sets of vertices $X \subseteq V = \cup_{v \in X} N_G(v)$

The **degree** of a vertex v is the **number** of vertices joined to v. Formally: $d_G(v) = |N_G(v)|$

Theorem: If G has an Euler walk, then either:

- Every vertex of G has even degree or
- All but two vertices v_0 and v_k have even degree, and any Euler walk must have v_0 and v_k as endpoints.

Does every single graph that satisfies both of these conditions have an Euler walk? No, because the graphs need to be **connected**.

Within a graph, we also have subgraphs and induced subgraphs. A **subgraph** $H = (V_H, E_H)$ of G is a graph with $V_H \subseteq$ and $E_H \subseteq E$. H is an **induced subgraph** if $V_H \subseteq V$ and $E_H = \{e \in E : e \subseteq V_H\}$.

For all vertex sets $X \subseteq V$, the graph is **induced** by X is:

$$G[X] = (X, \{e \in E : e \subseteq X\})$$

A **component** H of G is the maximal connected induced subgraph of G, so $H = G[V_H]$ is connected, but $G[V_H \cup \{v\}]$ is disconnected for all $v \in V \backslash V_H$.

Theorem: Let $G = (V, E)$ be a **connected** graph, and let $u, v \in V$. Then, G has an Euler walk from u to v if and only if either:

1. $u = v$ and every vertex of G has even degree, or
2. $u \neq v$ and every vertex of G has even degree except u and v.

3 Sequential Processes

4 Fast Fourier Transform

4.1 Polynomials

A **degree** $n-1$ polynomial in x can be seen as a function:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i$$

Any integer that's bigger than the degree of a polynomial is a *degree bound* of said polynomial. The polynomial A is:

$$a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 \cdots + a_{n-1}x^{n-1}$$

The values a_i are the *coefficients*, the degree is $n-1$ and n is a degree bound. We're able to express any integer as some kind of polynomial by setting x to some base, say for decimal numbers:

$$A = \sum_{i=0}^{n-1} a_i \cdot 10^i$$

The variable x just allows us to evaluate the polynomial at a point. A really fast way to evaluate the polynomial is to use **Horner's Rule**.

Horner's Rule

Instead of computing all the terms individually, we do:

$$A(3) = a_0 + 3 \cdot (a_1 + 3 \cdot (a_2 + \cdots + 3 \cdot (a_{n-1})))$$

This method requires $O(n)$ operations. For example, if we consider $A(x) = 2 + 3x + 1.x^2$, we can evaluate this as:

$$A(x) = 2 + x(3 + 1.x)$$

Once we have our polynomial representations, we might be doing some arithmetic with them. We're allowed to write polynomials in a *coefficient representation*. Here, the addition of $C = A+B$ constructs C as the vector:

$$(a_0 + b_0, a_1 + b_1, a_2 + b_2, \ldots, a_{n-1} + b_{n-1})$$

A and B should really have the same length, but we're allowed to just pad the coefficients with zero to make this the case.

4.1.1 Point value Representation

We know that if we're given a polynomial, we can graph it. We can use this fact to represent a polynomial as a list of its points. For point value representation, the addition $C = A + B$ constructs C as:

$$\{x_0, y_0 + z_0), (x_1, y_1 + z_1), (x_2, y_2 + z_2), \ldots, (x_{n-1}, y_{n-1} + z_{n-1})$$

where x_i is a point, $y_i = A(x_i) and z_i = B(x_i)$.

It's important to note that the two value representations **must** use the same evaluation points. Both of the operations are $O(n)$ in terms of how long they take.

4.1.2 Polynomial multiplication

Computing a polynomial multiplication (also called **convolution**) is a little harder than addition. It does, however, become much easier when we use point value representation:

$$\{x_0, y_0 \cdot z_0), (x_1, y_1 \cdot z_1), (x_2, y_2 \cdot z_2), \ldots, (x_{n-1}, y_{n-1} \cdot z_{n-1})$$

where x_i is a point, $y_i = A(x_i) and z_i = B(x_i)$.

The normal method of calculating multiplication is $O(n^2)$, while using point value representation only takes $O(n)$. So, is there an easy way to convert to point value representation? Actually, yes. What we need to do is to evaluate the polynomial to a point-value representation, multiply and finally interpolate (the opposite of evaluating) back again.

Long story short, we need to develop two fast algorithms that construct the coefficients for the point value representation and then interpolate. So the main steps to multiply the two polynomials A and B (of degree n) are:

1. *Double degree bound*: Create coefficient representations of $A(x)$ and $B(x)$ as degree bound $2n$ polynomials by adding n high-order zero coefficients to each.
2. *Evaluate*: Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of FFT of order $2n$
3. *Pointwise multiply*: compute a point-value representation of $cC(x) = A(x)B(x)$ by multiplying the values Pointwise
4. *Interpolate*: Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT.

The first and third steps are really easy to perform in $O(n)$ time. The claim is that if we evaluate at the complex roots of unity then we can perform steps 2 and 4 in $O(n \log n)$ time.

4.2 Evaluation at roots of unity

First, we need to look at evaluation. We need to evaluate a polynomial of degree n at n different points. It appears that the complexity of our method is just going to be $O(n^2)$, but there is a faster way of doing it than just using Horner's rule. This is when we evaluate the points at *special* points (namely the **N-th complex roots of unity**).

N-th complex roots of unity

- The roots of unity are the values $\omega_N = e^{2\pi ij/N}$ for $j = 0, 1, \ldots, N-1$
- Say we are evaluating at N pints, we take the N-th complex roots of unity ω_N
- This means we evaluate the polynomial at the points:

$$\omega_N^0, \omega_N^1, \omega_N^2, \ldots, \omega_N^{n-1}$$

We want to evaluate a polynomial A at the n roots of unity. Therefore, we evaluate:

$$A(x) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

for every $k = 0, 1, \ldots, n-1$

We define the vector of results of these evaluations as:

$$y_k = A(\omega_n^k)$$

This vector $y = (y_0, \ldots, y_{n-1})$ is the **discrete Fourier Transform** (DFT) of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$

The Cancellation Lemma

$$\omega_{dN}^{dk} = \omega_N^k$$

The Halving Lemma

If $N > 0$ is even then the squares of the N complex N-th roots of unity are the $N/2$ complex $N/2$-th roots of unity
The proof of this is that we have $(\omega_n^k)^2 = \omega_{n/2}^k$ for any nonnegative integer k because of the cancellation lemma.

4.3 Fast Fourier Transform

The really basic idea of the Fast Fourier transform is that we define two new polynomials;

$$A^{[0]}(x) = a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1}$$

5 Dynamic Programming

We use *dynamic programming* for finding efficient algorithms for problems that can be broken down into simpler, overlapping subproblems. Basically,

1. Find a recursive formula for the problem (in terms of answers to the subproblems)
2. Write down a naive recursive algorithm
3. Speed it up by storing the solutions to the subproblems (**memoization**)
4. Derive an iterative algorithm by solving the subproblems in good order.

5.1 Weighted interval scheduling

We've seen the scheduling problem before on the course, but this is different (and as it turns out, harder).

A **schedule** is a set of compatible intervals. The *weight* of a schedule is the sum of the weight of the intervals it contains.

We could use a greedy algorithm, but it is really slow and not what we're looking for. It's not even going to cut it full stop.

How is the input provided?

The intervals are given in an array A of length n.

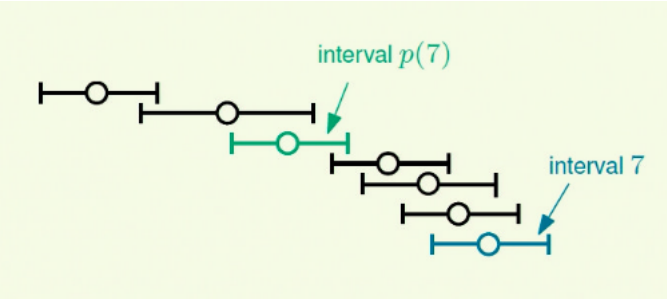
$A[i]$ stores a triple (s_i, f_i, w_i) which defines the ith interval.

The intervals are sorted by finish time i.e. $f_i \leq f_{i+1}$.=, or the interval i finishes before the interval $i+1$ finishes.

Compatible Intervals

For all i, we let $p(i)$ to be the rightmost interval (in order of finish time) which finishes before the ith interval but doesn't overlap it.

In the below example, if we take $i = 7$, $p(7) = 3$ because it's the first interval that doesn't overlap with it (and 3 is the position of the interval in the array).



$p(4) = 2$, and $p(2) = 0$ because there is no interval that exists. **Note** that we index from 1 because otherwise 0 would not make sense.

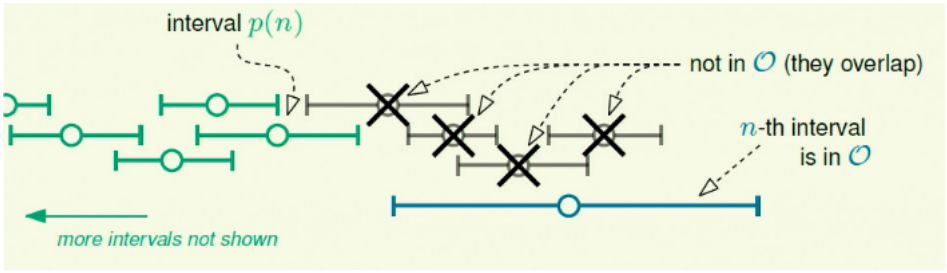
Claim. We can precompute all $p(i)$ in $O(n \log n)$ time. For now, we can assume that this is true, but I will revisit this and prove why. If you did it naively, it would be $O(n^2)$ time. Obviously it's something to do with dynamic programming (because that's what this section is about).

5.1.1 Finding a solution

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3, \ldots, n\}$ with weight OPT. Now, either the nth interval is in schedule $\mathcal{O}$ or it isn't.

Let's look at the case where it is not in $\mathcal{O}$. Schedule $\mathcal{O}$ is also an optimal schedule for the problem with the input consisting of the intervals $\{1, 2, 3, \ldots, n-1\}$ because n is not in it. Therefore, in this case, we have $OPT = OPT(n-1)$. (By the way, $OPT(i)$ is the weight of an optimal schedule for the intervals $\{1, 2, 3, \ldots, i\}$).

Now, let's consider the case that it **is** in $\mathcal{O}$. The only other intervals that could be in $\mathcal{O}$ are $\{1, \ldots, p(n)\}$. We know that everything larger than $p(n)$ is incompatible with the nth schedule.



Schedule $\mathcal{O}$ with interval n removed gives an optimal schedule for the intervals $\{1, 2, \ldots, p(n)\}$ so we now have that $OPT = OPT(p(n)) + w_n$.

To summarise:

- **Case 1**: The nth interval is not in $\mathcal{O}$, then $OPT = OPT(n-1)$
- **Case 2**: The nth interval is in $\mathcal{O}$, then $OPT = OPT(p(n)) + w_n$

So, which one do we choose? We choose the bigger one:

$$OPT = \max(OPT(n-1), OPT(p(n)) + w_n)$$

We just now replace all n with i to give us our final recursive algorithm

$$OPT = \max(OPT(i-1), OPT(p(i)) + w_i)$$

There's no easy way to find these solutions, you just have to kind of look at the problem for a while and then it will come to you.

5.1.2 Writing down the recursive algorithm

Now we have the formula, we get the recursive algorithm:

```
WIS(i)
    If (i=0)
        Return 0
    Return max(WIS(i-1),WIS(p(i)) + wi)
```

This algorithm is pretty much exponential given the wrong input. If $T(n)$ is the run time of WIS(n) using overlapping intervals, then $T(n) > 2T(n-1)$. This is $O(2^{n/2})$.

5.1.3 Memoization

To solve this, we need to store the solutions to the subproblems.

```
MEMWIS(i)
    If (i=0)
        Return 0
    If WIS[i] undefined
        WIS[i] = max(MEMWIS(i-1),MEMWIS(p(i)) + wi)
    Return WIS[i]
```

In this algorithm, we store the solutions to the previously computed subproblems in an n length array called WIS. The time complexity of computing MEMWIS(n) is now $O(n)$. Unfortunately, linear recursion is still kind of bad.

To compute a value in the array WIS[i], we need both WIS[$i-1$] and WIS[$p[i]$]. Both of these are to the left of WIS[i]. Therefore, we should go through this array from left to right. This is because every time we compute something, we already have all of the things we need. This gives us a new algorithm:

```
ITWIS(n)
    If (i=0)
        Return 0
    For i=1 to n
        WIS[i] = max(WIS[i-1], WIS[p(i)] + wi)
```

This is an iterative dynamic programming algorithm that runs in $O(n)$ time. The iterative method is better because if we use recursion in $O(n)$ time, we'll probably overload the stack, so this makes the memory footprint of the algorithm now much smaller. It **does** however mean that we need to precompute the $p(i)$ values.

5.1.4 Computing the p(i) function

Revised claim. We can precompute any $p(i)$ in $O(\log n)$ time. Remember that s_i is the start of interval i and f_i is the finish time of interval i. We want to find the unique value $j = p(i)$ such that:

$$f_j < s_i < f_{j+1}$$

Because the input is sorted by finish times, we can find j just by using binary search in $O(\log n)$ time. We can now precompute all $p(i)$ in $O(\log n)$ time.

6 Dynamic search structures

A dynamic search structure stores a set of elements. Each element x must have a unique key: `x.key`. The following operations are supported:

- `insert(x,k)`: inserts `x` with key `k = x.key`

6.1 Binary search trees

Remember that in a binary search tree:

- All the nodes in the left subtree have smaller keys
- All the nodes in the right subtree have larger keys

We perform a **`find`** operation by following a path from the root.

6.2 2-3-4 Trees

The idea of this tree is that the nodes can have anywhere between 2 and 4 children (*hence the name*).

Perfect balance means that every path from the root to a leaf has the same length. Always. Forever.

- **2-node**: 2 children and 1 key
- **3-node**: 3 children and 2 keys
- **4-node**: 4 children and 3 keys

Like in a binary search tree, the keys held at a node determine the contents of its subtrees.

Just like in a binary tree, we have a **`find`** operation. We perform the operation by following a path from the root. Decisions are made by inspecting the keys at the current node and following the appropriate edge.

The time complexity of the **`find`** operation is $O(h)$ again.

6.2.1 The insert operation

To perform `insert(x,k)`, we need to:

1. Search for the key `k` as if we were performing `find(k)`.
2. If the leaf is a 2-node, then we insert `(x,k)` and convert it into a 3-node.
3. If the leaf is a 3-node, then we insert `(x,k)` and convert it into a 4-node.
4. If the leaf is a 4-node, we just make sure it never happens.

6.2.2 Splitting 4-nodes

We can `split` any 4-node into two 2-nodes *if* its parent isn't a 4-node. For example:

We push the extra key up to the parent (which wouldn't work if the parent is a 4-node). The subtrees haven't changed size, so the path lengths have not changed due to this `split` operation. Therefore, if it was **perfectly balanced** before, then it's perfectly balanced after.

`splitting` the root increases the height of the tree and increases the length of all the root-leaf paths by one so it maintains the **perfect balance** property. This is the only way that `insert` can affect the lengths of the paths.

Each `split` takes $O(1)$ time, so overall `insert` takes $O(\log n)$ time.

6.2.3 Other operations

`Fuse` is an operation that combines two 2-nodes (with the same parent) into a 4-node (provided the parent isn't a 2-node).

`Transferring` keys is possible. If there is a 2-node and a 3-node, we can perform a `Transfer` (even if the parent is the root). No path lengths change from this operation and the time taken is $O(1)$ time.

6.2.4 Deleting a node

To perform `Delete(k)` on a **leaf**, we:

1. Search for the key `k` using `find(k)`. We use `fuse` and `transfer` to convert the 2-nodes as we go down.
2. If the leaf is a 3-node, we delete `(x,k)`

If we `fuse` the root, then the height can decrease by 1. This is the only time the tree can decrease in height.

If we want to `delete` something other than a leaf, we need to:

1. Find the **predecessor** of `k` (this is the same as `find`). This is the element with the largest key k' such that $k' < k$
2. Call `delete(k')`. Fortunately, `k'` is always a leaf
3. Overwrite `k` with another copy of `k'`.

6.3 Summary

A 2-3-4 tree is a data structure that's based on a tree structure. They are a little awkward to implement because all nodes don't have the same number of children. So in practice, we use something called a **red-black** tree. It is similar to a binary tree and supports `insert`, `find`, `delete` functions.

Why do we bother learning these 2-3-4 trees then? Well, they're a little bit nicer and less complicated to think about and they're basically the same structure.

7 Making shortcuts

What happens if we add some shortcuts to a set of nodes. Because in the real world it's possible to take shortcuts in terms of a train line or something similar.

We might attach a second linked list containing only some of the keys. How do we now perform `find(k)` in this linked list?

To perform `find` we start in the top list and go right until we come to a key $k' > k$. Then, we move down to the bottom list and go right until we find k. Imagine that we decide to place m keys in the top list, and the bottom list contains n keys (this will always be true since the bottom list has all of the keys). Where should we put the m keys to minimise the *worst* case (for a find operation)?

If we spread out the m keys evenly, we get the biggest overall improvement. If we do an uneven spread, some of the keys will be found really fast, but there will be some poor cases, and since we're trying to minimise the worst case, we want to make sure all cases are catered for. Now, the worst case time for a `find` operation becomes $O(m + n/m)$.

By setting $m = \sqrt{n}$, we get the **worst case** time for a `find` operation to be $O(\sqrt{n})$.

7.1 More levels

We've looked at having 2 lists, but what if we have more than that? The more levels we introduce, the better it gets. Each *level* will contain **half** of the keys (called rounding up) from the level below. They are chosen to be spread as evenly as possible.

The bottom level still contains every key, and every level contains the leftmost and rightmost keys.

since each level has half of the keys from the below level, there are $O(\log n)$ levels (kinda like a tree).

8 Line segments

Consider n line segments. Find all of the intersections. How do we go about this?

The simplest algorithm is to test every pair of line segments.

Let s_i denote the i-th line intersection:

```
for i = 1,2, ..., n
    for j = 1,2, ...,n
        if (s[i] intersects s[j]) and (i != j)
            output (i,j)
```

Given two line segments s_i and s_j, described by their end point coordinates, we can decide whether (and also where) they intersect in $O(1)$ time. This can be proved because any computation on two objects with $O(1)$ (constant) space descriptions take $O(1)$ time. It doesn't tell you how to do it, but just know that this is true.

The algorithm above, however, runs in $O(n^2)$ time (because checking each pair of lines takes $O(1)$ time).

If there are n line segments we could have a maximum of $(n/2)^2$ intersections. If we want to output all of the intersections, then we can't expect to do better than $O(n^2)$ in the worst case simply because the time complexity of printing the output.

8.1 Output sensitive algorithms

Up to this point in the course, we've only looked at how the actual input affects the runtime. With this algorithm, however, we also need to consider the size of the output.

Eventually, we'll work up to an algorithm with runtime $O(n \log n + k \log n)$. When K is really big, then this algorithm is actually worse than the naive algorithm with $O(n^2 \log n)$. But, when k is small, the algorithm is much better with $O(n \log n)$.

8.2 Simplifying restrictions

To make things a bit easier, we are not allowing:

- Horizontal line segments
- Two end points with the same
- Three or more line segments that intersect at the same point
- Overlapping line segments

8.3 First observation

The y**-span** of s_i is the range of the y axis that s_i takes. From this definition, if s_i and s_j don't have overlapping y-spans, they don't intersect. This method suggests an overall approach called *line sweeping*. We sweep a horizontal line through the plane from top to bottom and find intersections as we go.

When we see that there are two lines in the same y-span that are also next to each other, we say that the two lines are *adjacent* **at this y-coordinate**. Some pair of lines that are adjacent at some y coordinate may not be adjacent at another because another line might come between them.

Note! Two line segments that are *never* adjacent **cannot intersect**. Intuitively, if they are never actually next to each other, they cannot cross.

8.3.1 Event points

The **event points** are the 'interesting positions' of a line, which we define as the end points of the line and the intersection points. We have to detect the intersection points on the fly **before** we get to them.

The total number of event points is $O(n+k)$ where n is the number of endpoints and k is the number of intersections.

8.3.2 Status of the sweep line

The status of the sweep line is the set of line segments that currently intersect the sweep line. We order this from *left to right* by where they intersect.

The status of the sweep line can only change at event points.

We will store the status of the sweep line in a data structure. This allows efficient updates.

8.3.3 Updating the sweep line

Every time the sweep line moves to the next event point, we update the status data structure. If the event point is the *top of a line segment*, we insert it into the status data structure at the appropriate place (using the insert function).

Then, we check whether the segment will intersect either of the adjacent segments.

If the event point is the *bottom of a line segment* then we delete it from the status data structure.

Finally, if the event point is an *intersection point*, we **swap** the two line segments in the data structure.

9 Shortest Path Revisited

We've already looked at the shortest path thing, but now, we're going to do it with negative weights. Bellman-Ford's algorithm solves the **single source shortest paths** problem (in a weighted directed graph).

It finds the shortest path from a given *source* vertex to every other vertex. The weights are allowed to be both positive and negative, and the graph is stored as an **adjacency list**.

9.1 Negative weight cycles

If some of the edges in the graph have negative weights, the idea of a shortest path might not make sense:

A negative weight cycle is a path from a vertex v back to v such that the sum of the edge weights is negative. If there is a path from s to t, which includes a negative weight cycle, there is no shortest path from s to t.

Firstly, we are going to discuss a simpler version of the Bellman-Ford algorithm, that assumes there are **no such cycles**.

The algorithm repeatedly asks, for every edge (u, v) "can I find a shorter route to v if I go via u?". Here is the algorithm:

```
sub MostOfBellman-Ford
    for all v, set dist(v) = infinity
    set dist(s) = 0
    for i = 1 to |V|
        for each edge (u,v) in E
            if dist(v) > dist(u) + weight(u,v) then
                dist(v) = dist(u) + weight(u,v)
            end if
        next
    next
```

```
end sub
```

This algorithm runs $|V|$ iterations. In each iteration, we relax **every** edge (u, v).

Now, consider the algorithm where in each algorithm you relax every edge (instead of only one). After enough iterations, we get all edges to be the shortest path. But, how many iterations is enough? Well, we need to do one iteration for each node in the path we want to find the shortest path of.

Due to the fact that we have no negative weight cycles, deleting a cycle from a path **cannot** increase its length. Therefore, there is a shortest path between two nodes containing no cycles. Therefore, we have proved that 'enough' cycles is $|V|$.

Of course, if there is no path between the two nodes, then at termination, `dist(v)= infinity`. If there is a shortest path between two nodes, then it contains at most $|V|$ edges (assuming there are no negative weight cycles).

So, what's the rest of the algorithm?

9.2 The rest of the algorithm

```
sub bellman-ford
    for all v, set dist(v) = infinity
    set dist(s) = 0
    for i = 1 to |V|
        for each edge (u,v) in E
            if dist(v) > dist(u) + weight(u,v) then
                dist(v) = dist(u) + weight(u,v)
            end if
        next
    next

    for each edge (u,v) in E
        if dist(v) > dist(u) + weight(u,v)
            print("negative weight cycle found")
        end if
    next
end sub
```

The final check outputs a message if there is a negative weight cycle, and it does this by checking if there is a path that is shorter than the shortest path (that is a contradiction) and therefore there must be a negative weight cycle.

9.3 All-pairs Shortest Paths

In previous lectures, we have focused on the single source graphs, But, what if we want to find the shortest paths from every vertex to every other vertex?

- If the graph has **non-negative edge weights**, then we can just run Dijkstra's algorithm $|V|$ times. ($O(|V||E|\log|V|)$ time).
- If the graph has both **positive and negative edge weights**, we can repeatedly run Bellman-Ford's algorithm. ($O(|V|^2|E|)$ time)

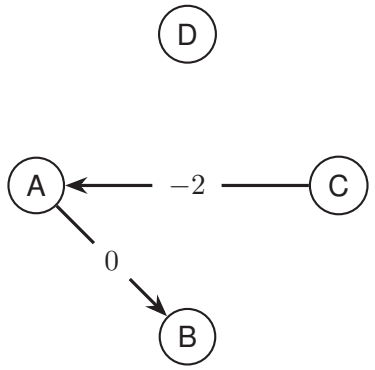
The output contains the length of the shortest path between every pair of vertices, so there are $|V| \cdot (|V| - 1)$, therefore we can't expect to do better than $O(|V|^2)$, based on pairs alone.

Now, imagine that $|E| \approx \frac{|V|^2}{4}$ (basically ,each vertex has an edge to about half of the other vertices). This is quite a dense graph. If V was 10000, then this becomes just too big.

If the graph is really sparse ($|E| \approx 5|V|$), (basically each vertex has an edge to about 5 other vertices).

9.3.1 Johnson's algorithm

We've already seen one algorithm for all-pairs shortest paths that takes $O(|V||E|\log|V|)$ time.If the graph had non-negative weights, we can just repeatedly apply Dijkstra's algorithm. However, we are interested in graphs that have both positive and negative edge weights. The approach employed by Johnson's algorithm is to reweight the edges so that the resulting graph has non-negative edge weights, and *then* repeatedly runs Dijkstra's algorithm. For example;



9.4 Reweighted paths

Let the function h give a value $h(v)$ for each vertex $v \in V$ change the weight of every edge (u, v) to be:

$$\text{weight}'(u, v) = \text{weight}(u, v) + h(u) - h(v)$$

Lemma: Any path is a shortest path in the original graph if and only if it is a shortest path in the reweighted graph.

In summary, to reweight the graph, we let the function h give a value $h(v)$ for each vertex $v \in V$. Then, we change the weight of every edge (u, v) to be weight$'(u, v) =$ weight$(u, v) + h(u) - h(v)$.

Lemma: Any path is a shortest path in the original graph if and only if it is a shortest path in the reweighted graph.

Fact: If ℓ is the length of a path from u to v

9.5 Choosing h

We first add one additional vertex called s to the original graph. We also add an edge (s, v) from s to each other vertex $v \in V$ (each of the edges has weight 0). For each v, let $\delta(s, v)$ denote the length of the shortest path from s to v. We then define $h(v)$ to equal $\delta(s, v)$.

Consider any edge $(u, v) \in E$. The key observation is that $\delta(s, v) \leq \delta(s, u) +$ weight(u, v). Rearranging this, we end up with weight$'(u, v) =$ weight$(u, v) + h(u) - h(v)$.

9.6 Johnson's actual algorithm

We can now piece together Johnson's algorithm which operates as follows:

1. Add one additional vertex called s to the original graph
2. For each vertex, add an edge (s, v) with weight 0
3. Run the Bellman-Ford algorithm with source s. (This calculates the shortest path lengths $\delta(s, v)$) for all v.
4. Reweight each edge $(u, v) \in E$ so that

$$\text{weight}'(u, v) = \text{weight}(u, v) + h(u) - h(v)$$

5. For each vertex u, run Dijkstra's algorithm with source $s = u$.
6. For each pair of vertices u, v compute

$$\delta(u, v) = \delta'(u, v) + h(v) - h(u)$$

Therefore, the overall time complexity is $O(|V||E| \log |V|)$, which is actually the same time period as Dijkstra's algorithm.