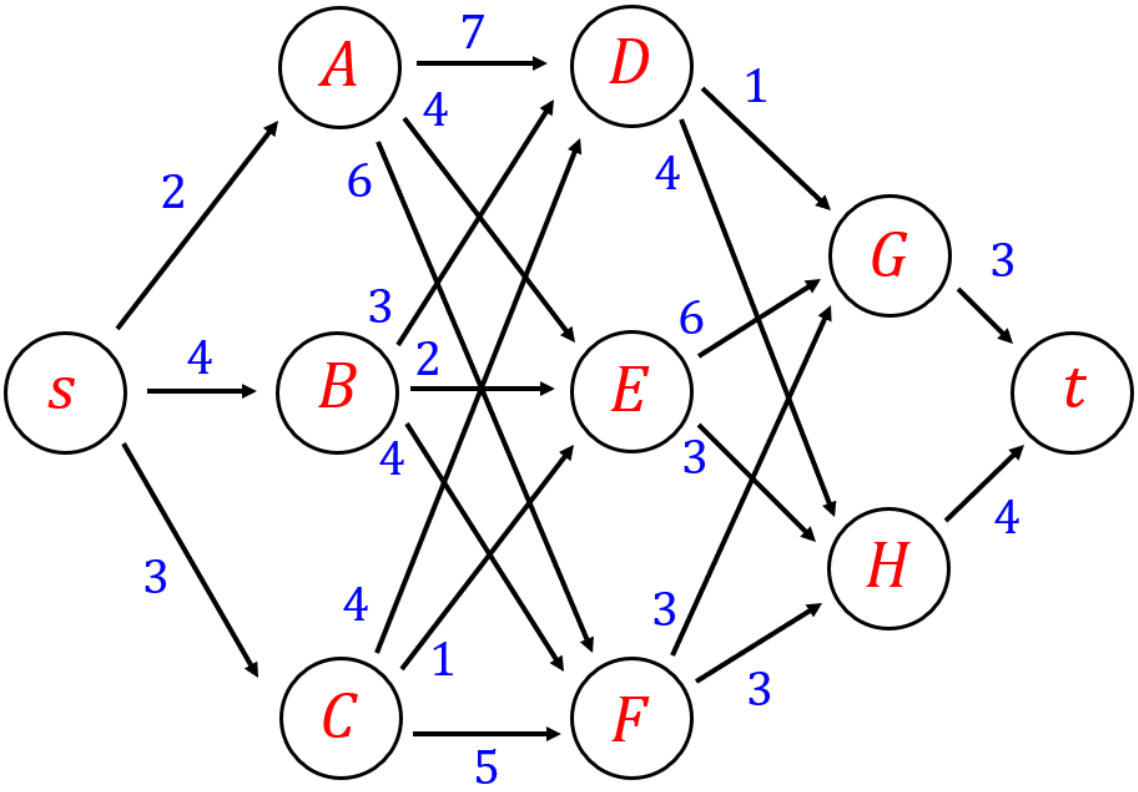
LECTURE 17: DYNAMIC PROGRAMMING

Today: A cool way of solving a large class of programming problems

1. Cheapest Flight Problem

Example 1:

Consider the following network



Date: Thursday, November 3, 2022.

(It looks confusing at first but just think of)

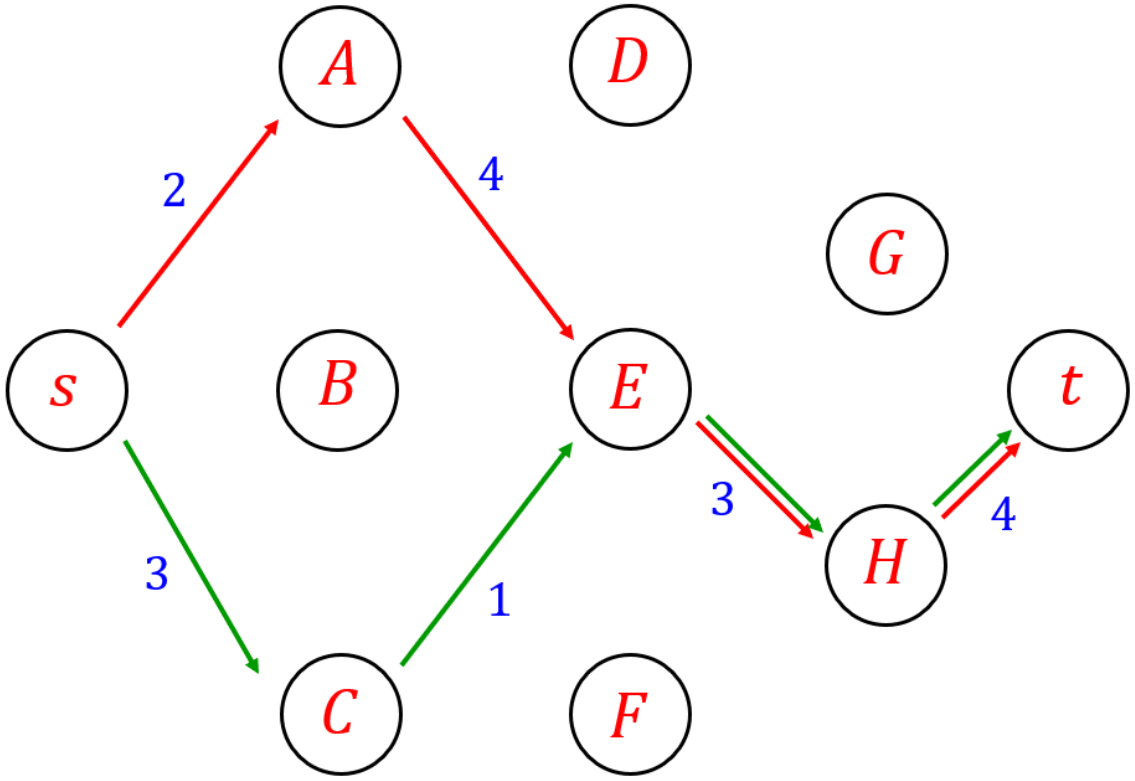
Here the vertices are cities and edges are the **cost** of each flight.

Goal: Find the cheapest flight from s (Seattle) to t (Tampa)

Greedy Algorithm: Just pick the cheapest flight at each step!

This gives you the path $s \to A \to E \to H \to t$ (in red) with total cost

$$2 + 4 + 3 + 4 = 13$$



However this is not optimal!

Better solution: If you take the path $s \to C \to E \to H \to t$ (green) then the total cost is

$$3 + 1 + 3 + 4 = 11$$

The reason the greedy algorithm doesn't work is because, while flying from s to A might be cheap, you're missing out on the much cheaper option at C.

2. Dynamic Programming

The idea of dynamic programming is to deal with this problem recursively, by solving smaller sub-problems of the same nature.

Note: You already saw an example of recursion, where you cut a leaf from a tree and got a smaller tree

Analogy: Suppose you have a friend f who works perfectly and can book you the best flight up to a certain city.

Definition:

$$f(x) = \text{ Best flight (total cost) from } s \text{ to } x$$

$$\begin{aligned} f(s) &= 0 \\ f(a) &= \text{best flight from } s \text{ to } a \ = 2 \\ f(t) &= \text{best flight from } s \text{ to } t \ = \ \text{our Goal} \end{aligned}$$

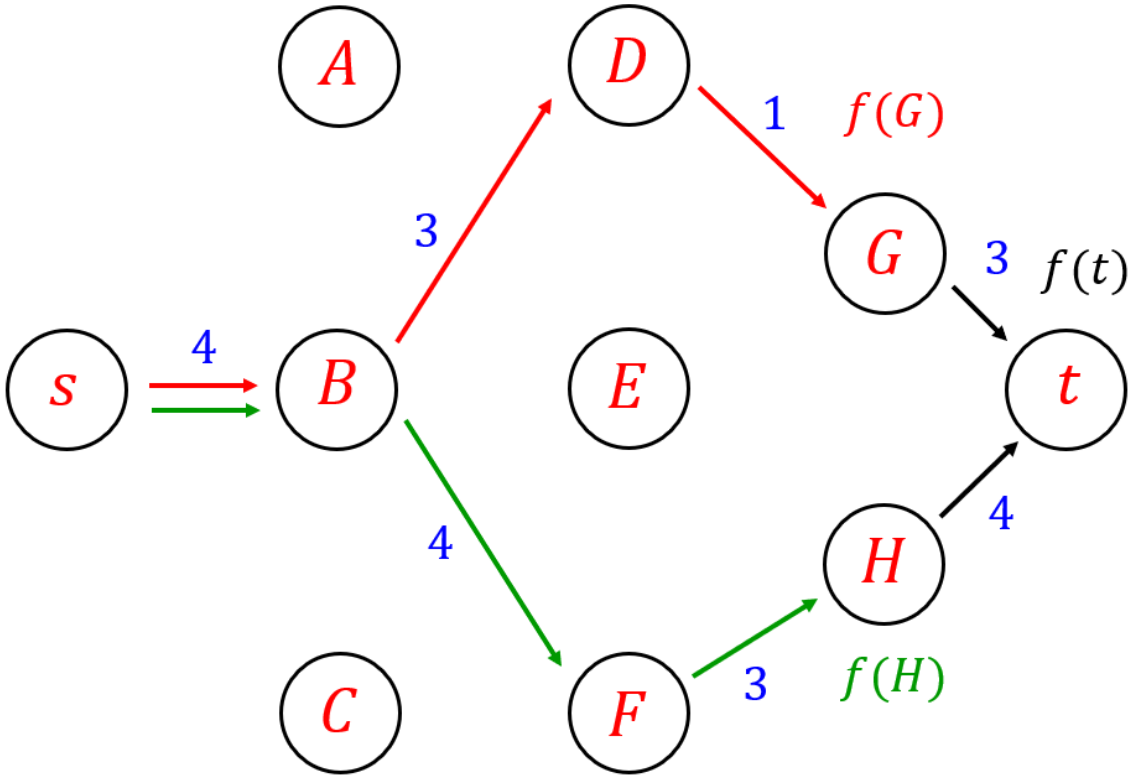
Now let's think about this backwards:

In order to find $f(t)$, the best ticket from s to t:

- Either your friend books you a ticket to G (this is $f(G)$) and you pay 3 for $G \to t$
- Or your friend books you a ticket to H (this is $f(H)$) and you pay 4 for $H \to t$

This gives two tickets and you choose the cheapest one. In other words:

$$f(t) = \min\{f(G) + 3, f(H) + 4\}$$



Notice how **recursive** this is, you write f in terms of f.

But now we can continue! How do we find $f(G)$? We fly optimally to either D, E, F and pick the cheapest option to fly from there to G, that is

$$f(G) = \min\{f(D)+1, f(E)+3, f(F)+3\}$$

And so we have a bunch of equations for f,

$$\begin{aligned} f(t) &= \min\{f(G)+3, f(H)+4\} \\ f(G) &= \min\{f(D)+1, f(E)+6, f(F)+3\} \\ f(H) &= \min\{f(D)+4, f(E)+3, f(F)+3\} \\ f(D) &= \cdots \\ &\vdots \\ f(s) &= 0 \end{aligned}$$

Although this works, this very quickly gets out of hands, especially for graphs with millions of nodes! This is why it's useful to have an ordering on the vertices.

3. Ordering on Vertices

Notice that in the graph above, G "comes after" D. This is because, in terms of our tasks, we need $f(D)$ to calculate $f(G)$ but we don't need $f(G)$ to calculate $f(D)$.

Definition:

We say $j > i$ if $f(j)$ requires us to calculate $f(i)$

So in this scenario, we have $G > D, G > E, G > F$ and but also things like $G > A$.

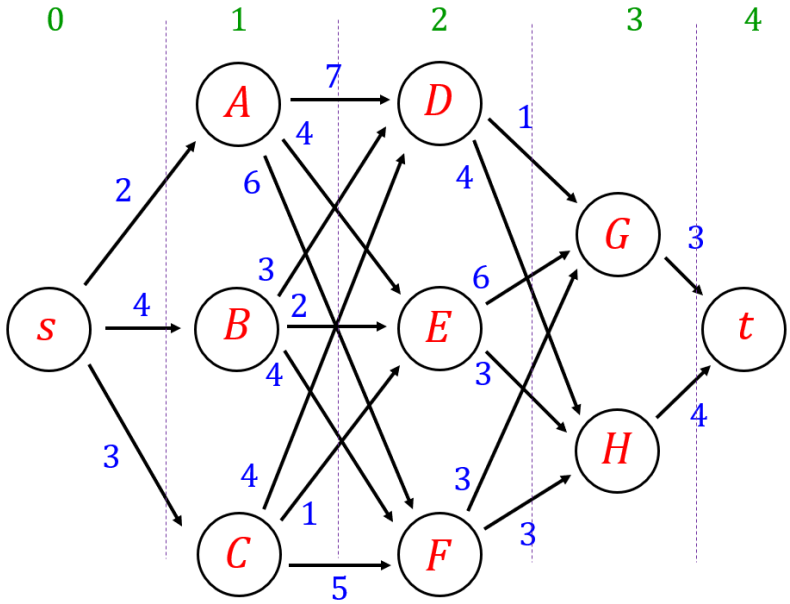
Notice that not all vertices can be compares, for example we can't compare G and H

This is called a **partial order** on the vertices[1]

Note: This allows us to define an orientation of the graph, where we say $i \to j$ if $j > i$. In this example it coincides with our usual orientation, but it doesn't have to.

What's nice about this is that this new oriented graph doesn't have any cycles[2], sometimes called a **directed acyclic graph** (DAG)

More importantly, this allows us to sub-divide the graph into stages, where the last stage (stage 4) is t, the stage before (stage 3) is all the vertices smaller than t (so here G and H), the stage before (stage 2) is all the vertices smaller than G and H (so D, E, F) etc. and the very first stage (stage 0) is s.



[1]Partial ordering means $i \not> i$, and $i > j \Rightarrow j \not> i$ and finally $k > j$ and $j > i$ implies $k > i$

[2]follows from the third property: If $i_1 > i_2 > \cdots > i_n > i_1$ (cycle) then we'd have $i_1 > i_1$

4. Implementation

With this ordering at hand, let's see how to implement dynamic programming to solve our problem. Once again, we want to work backwards, starting at t

STAGE 4: t

Departure \ Arrival	t	Smallest Cost	City
G	3	3	$G \to t$
H	4	4	$H \to t$

STAGE 3: G, H

Departure \ Arrival	$G \to t$	$H \to t$	Smallest Cost	City
D	$1 + 3 = 4$	$4 + 4 = 8$	4	$D \to G$
E	$6 + 3 = 9$	$3 + 4 = 7$	7	$E \to H$
F	$3 + 3 = 6$	$3 + 4 = 7$	6	$F \to G$

(For the first entry, we found the cost of $D \to G$ (1) and added it to the cost of $G \to t$ (3) from the previous table)

STAGE 2: D, E, F

Dep. \ Arr.	$D \to t$	$E \to t$	$F \to t$	Cost	City
A	$7 + 4 = 11$	$4 + 7 = 11$	$6 + 6 = 12$	11	$A \to D$ or E
B	$3 + 4 = 7$	$2 + 7 = 9$	$4 + 6 = 10$	7	$B \to D$
C	$4 + 4 = 8$	$1 + 7 = 8$	$5 + 6 = 11$	8	$C \to D$ or E

(For the first entry, we found the cost of $A \to D$ (7) and added it to the smallest cost of $D \to t$ (4) from the previous table)

STAGE 1: A, B, C

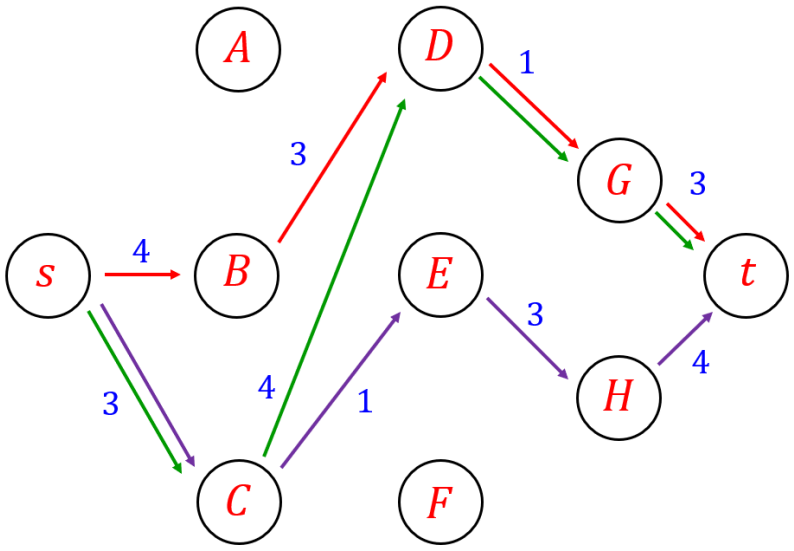
Arr. / Dep.	$A \to t$	$B \to t$	$C \to t$	Cost	City
s	$2 + 11 = 13$	$4 + 7 = 11$	$3 + 8 = 11$	11	$s \to B$ or C

(For the first entry, we found the cost of $s \to A$ (2) and added it to the smallest cost of $A \to t$ (11) from the previous table)

Optimal Value: 11

Optimal Paths:

$$s \to B \to D \to G \to t$$
$$s \to C \to D \to G \to t$$
$$s \to C \to E \to H \to t$$



5. Application: Distance between Words

This is a fun application of dynamic programming, explaining roughly how autocorrect on your phone works.

For example, "SUNNING" gets autocorrected to "STUNNING" because "STUNNING" is in the phone's dictionary, and "SUNNING" is only one letter away.

Given: Two words, like SUNNY and SNOWY

Allowable Moves:

- **Insertion:** Add a letter to a word, like SUNNY $\to$ S**T**UNNY
- **Deletion:** Remove a letter from a word, like SUNNY $\to$ UNNY
- **Replacement:** Replace a letter, like **S**UNNY $\to$ **F**UNNY

Definition:

The **edit distance** is the minimum amount of edits to transform one word to another

Example 2:

Find the edit distance between SUNNY and SNOWY

The edit distance between SUNNY and SNOWY is 3 because one possible transformation is

$$\text{SUNNY} \to \text{S}\mathbf{N}\text{NNY} \to \text{SN}\mathbf{O}\text{NY} \to \text{SNO}\mathbf{W}\text{Y}$$

And you can't transform one word into another with 2 moves only.

Goal: Find the edit distance between two words:

Notation:

Given two words $x[1, 2, \cdots, m]$ and $y[1, 2, \cdots, n]$, where m, n are the lengths of x and y

Here $x[1, 2, \cdots, 5] =$ SUNNY and $y[1, 2, \cdots, 5] =$ SNOWY

In particular, we can define **truncated words** $x[1, \cdots, i]$ and $y[1, \cdots, j]$

For example, if $i = 2$ and $j = 3$ then

$$x[1, 2] = \text{ SU and } y[1, 2, 3] = \text{ SNO}$$

Edit Distance:

$E(i, j) =$ Edit Distance between truncations $x[1, \cdots, i]$ and $y[1, \cdots, j]$

In this example, $E(2, 3) = 2$ (SU $\rightarrow$ S**N**U $\rightarrow$ SN**O**)

Goal: Calculate $E(m, n) = E(5, 5)$

In order to use dynamic programming, we have to imagine again a friend who can do the task perfectly.

Trick: Consider the last letter $x[i] = U$ of the first word, here U. For the record, the last letter of the second word is $y[j] = O$.

Then you can do either of three things:

(1) Either delete $x[i] = U$ from $x[1, \cdots, i]$, so SU $\rightarrow$ S

(2) Or insert $y[j]$ to $x[1, \cdots, i]$, so SU $\rightarrow$ SUO

(3) Or replace $x[i]$ with $y[j]$, so SU $\rightarrow$ SN

(Here we pick the third option, but we don't even need to know that, that's the beauty of it)

In option 1, we did 1 move, and are left to compare $x[1, \cdots, i-1]$ with $y[1, \cdots, j]$ (since we already used the last letter of the x part)

In option 2, we did 1 move, and are left to compare $x[1, \cdots, i]$ with $y[1, \cdots, j-1]$ (since we already used the last letter of the y part)

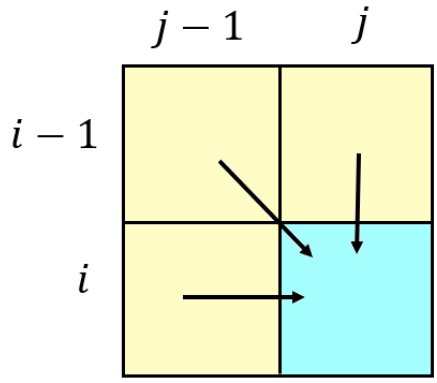
In option 3, we did 0 or 1 moves (0 if both letters coincide, like swapping S with S) and are left to compare $x[1, \cdots, i-1]$ and $y[1, \cdots, j-1]$ (we used the last letters of both parts)

Dynamic Programming Formulation:

$$E(i,j) = \min\{1 + E(i-1,j), 1 + E(i,j-1), (0 \text{ or } 1) + E(i-1,j-1)\}$$

And then you let the algorithm run its course, until you hit the starting step $i = 0$, in which case $E(0,j) = j$, or $j = 0$, in which case $E(i,0) = i$.

Note: Here $E(i,j)$ depends only on its immediate neighbors $E(i-1,j)$, $E(i,j-1)$ and $E(i-1,j-1)$, as in the following picture



Example 3:

Edit distance between POLYNOMIAL and EXPONENTIAL

If you implement the algorithm above, you'll find the edit distance is 6. The table below illustrates all the sub-calculations that are done

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6