Lecture 6: Dynamic Programming on Strings and Bellman-Ford Algorithm

Yury Makarychev

DP on Strings

Longest Common Subsequence

Subsequence of a String

A subsequence of string $s_1 \dots s_m$ is a string of the form $s_{i_1} \dots s_{i_k}$ where

$$i_1 < i_2 < \cdots < i_k$$

string: The University of Chicago

subsequence: hvrsag

(Empty string Λ is a subsequence of every string.)

cf. a substring is a subsequence formed by consecutive characters: rsity of Chi

Longest Common Subsequence

LCS: given two strings $s_1, \dots, s_m$ and $t_1, \dots, t_n$

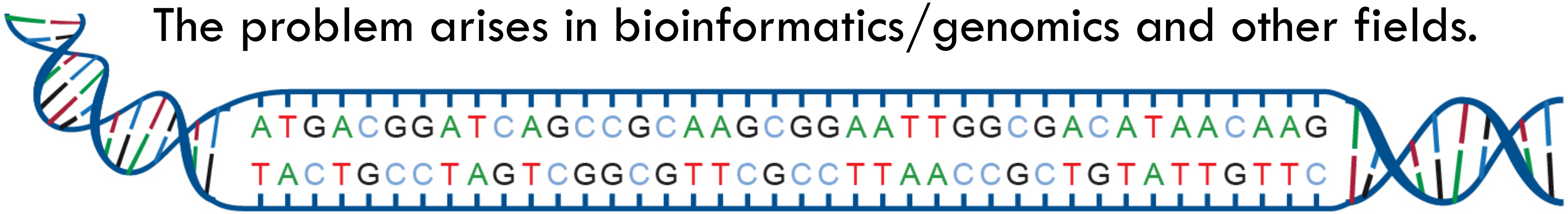
Find the longest common subsequence.

$s =$ "humpty_dumpty_sat_on_a_wall,_humpty_dumpty_had_a_great_ fall."

$t =$ "all_the_king's_horses_and_all_the_king's_men_couldn't_put_humpty_together_again."

$LCS(s,t) =$"_t_s_o_a_all_hupt_umpty_h_aga."

The problem arises in bioinformatics/genomics and other fields.



The image is courtesy of NHGRI.

DP for LCS

- ## Subproblems

subproblem (i, j): find the LCS of $s_1 \ldots s_i$ and $t_1 \ldots t_j$.

- ## DP Table

$T[0{:}m, 0{:}n]$. Entry $T[i, j]$ stores the optimal value of subproblem (i, j).

Our ultimate goal is to compute $T[m, n]$.

- ## Initialization

$T[0,*] = T[*, 0] = 0$

Recurrence

Matching characters

“humpty_dumpty_sat_on_a_wall,_humpty_dumpty_had_a_great_ fall.”

...

“all_the_king’s_horses_and_all_the_king’s_men_couldn’t_put_humpty_together_again.”

Let $s_{i_1} \ldots s_{i_k}$ and $t_{j_1} \ldots t_{j_k}$ be occurrences of LCS in s and t, respectively.

$$s_{i_1} \ldots s_{i_k} = t_{j_1} \ldots t_{j_k} \text{ where } k = LCS(s,t)$$

We say that s_{i_a} is matched with t_{j_a}.

Recurrence

Matching characters

“humpty_dumpty_sat_on_a_wall,_humpty_dumpty_had_a_great_ fall.”

...

“all_the_king’s_horses_and_all_the_king’s_men_couldn’t_put_humpty_together_again.”

- The matching is monotone (visually: the arrows above don’t intersect).
- $LCS(i, j)$ equals the size of the matching

Recurrence

Consider subproblem (i, j). We have the following options.

1. s_i is not matched
2. t_j is not matched
3. s_i is matched and t_j is matched

Q: Are these options mutually exclusive?

Q: Is it possible that none of them happens?

Recurrence

Option 1: s_i is not matched

$$s_1 s_2 s_3 \ldots s_{i-1} \boxed{s_i}$$
$$t_1 t_2 t_3 \ldots t_{j-1} t_j$$

$T[i, j] = ?$

Recurrence

Option 1: s_i is not matched

$$s_1 s_2 s_3 \ldots s_{i-1} \boxed{s_i}$$
$$t_1 t_2 t_3 \ldots t_{j-1} t_j$$

$$T[i, j] = T[i-1, j]$$

Recurrence

Option 2: t_j is not matched

$$s_1 s_2 s_3 \ldots s_{i-1} s_i$$
$$t_1 t_2 t_3 \ldots t_{j-1} \boxed{t_j}$$

$$T[i,j] = T[i,j-1]$$

Recurrence

Option 3: s_i is matched and t_j is matched

$$s_1 s_2 s_3 \ldots s_{i-1} \boxed{s_i}$$

$$t_1 t_2 t_3 \ldots t_{j'} \ldots \boxed{t_j}$$

What characters are s_i and t_j matched to?

Can s_i be matched with $t_{j'}$ with $j' \neq j$?

Recurrence

Option 3: s_i is matched and t_j is matched

$$s_1 s_2 \ldots s_{i'} \ldots s_{i-1} s_i$$

$$t_1 t_2 t_3 \ldots t_{j'} \ldots t_j$$

What characters are s_i and t_j matched to?

Can s_i be matched with $t_{j'}$ with $j' \neq j$?

No! Then t_j would be matched with some $s_{i'}$. The matching is not monotone!

Recurrence

Option 3: s_i is matched and t_j is matched

$$s_1 s_2 \dots s_{i'} \dots s_{i-1} s_i$$

$$t_1 t_2 t_3 \dots t_{j'} \dots t_j$$

➢ s_i is matched with t_j

Thus, $s_i = t_j$.

➢ Option 3 is possible only if $s_i = t_j$.

Recurrence

Option 3 is possible only if $s_i = t_j$.

We take the LCS for $s_1 \dots s_{i-1}$ and $t_1 \dots t_{j-1}$ and add one character $s_i = t_j$ to it.
(We use that $s_i = t_j$!)

- $T[i,j] = T[i-1, j-1] + 1$

Recurrence

If $s_i \neq t_j$ then

$$T[i,j] = \max(T[i-1,j], T[i,j-1])$$

If $s_i = t_j$ then

$$T[i,j] = \max(T[i-1,j], T[i,j-1], T[i-1,j-1]+1)$$

Recurrence

If $s_i \neq t_j$ then

$$T[i,j] = \max(T[i-1,j], T[i,j-1])$$

If $s_i = t_j$ then

$$T[i,j] = \max(T[i-1,j], T[i,j-1], T[i-1,j-1]+1)$$

In fact, we can always match s_i with t_j if $s_i = t_j$

$$s_1 s_2 s_3 \ldots s_{i-1} s_i \qquad\qquad s_1 s_2 s_3 \ldots s_{i-1} s_i$$

$$t_1 t_2 t_3 \ldots t_{j'} \ldots t_j \qquad\qquad t_1 t_2 t_3 \ldots t_{j'} \ldots t_j$$

Recurrence

If $s_i \neq t_j$ then

$$T[i,j] = \max(T[i-1,j], T[i,j-1])$$

If $s_i = t_j$ then

$$T[i,j] = T[i-1,j-1] + 1$$

$$\text{If } s_i \neq t_j \text{ then } T[i,j] = \max(T[i-1,j], T[i,j-1])$$
$$\text{If } s_i = t_j \text{ then } T[i,j] = T[i-1,j-1] + 1$$

Example

1	2	3	4	5	6
t	r	a	i	n	
s	t	r	o	n	g

6						
5						
4						
3						
2						
1						
0						
	0	1	2	3	4	5

$$\text{If } s_i \neq t_j \text{ then } T[i,j] = \max(T[i-1,j], T[i,j-1])$$
$$\text{If } s_i = t_j \text{ then } T[i,j] = T[i-1,j-1] + 1$$

Example

1	2	3	4	5	6
t	r	a	i	n	
s	t	r	o	n	g

6	0					
5	0					
4	0					
3	0					
2	0					
1	0					
0	0	0	0	0	0	0
	0	1	2	3	4	5

$$\text{If } s_i \neq t_j \text{ then } T[i,j] = \max(T[i-1,j], T[i,j-1])$$

$$\text{If } s_i = t_j \text{ then } T[i,j] = T[i-1,j-1] + 1$$

Example

1	2	3	4	5	6
t	r	a	i	n	
s	t	r	o	n	g

6	0					
5	0					
4	0					
3	0					
2	0					
1	0	0				
0	0	0	0	0	0	0
	0	1	2	3	4	5

Example

If $s_i \neq t_j$ then $T[i,j] = \max(T[i-1,j], T[i,j-1])$

If $s_i = t_j$ then $T[i,j] = T[i-1,j-1] + 1$

1	2	3	4	5	6
t	r	a	i	n	
s	t	r	o	n	g

6	0					
5	0					
4	0					
3	0					
2	0	1				
1	0	0				
0	0	0	0	0	0	0
	0	1	2	3	4	5

$$\text{If } s_i \neq t_j \text{ then } T[i,j] = \max(T[i-1,j], T[i,j-1])$$
$$\text{If } s_i = t_j \text{ then } T[i,j] = T[i-1,j-1] + 1$$

Example

1	2	3	4	5	6
t	r	a	i	n	
s	t	r	o	n	g

6	0	1				
5	0	1				
4	0	1				
3	0	1	2			
2	0	1	1			
1	0	0	0			
0	0	0	0			
	0	1	2	3	4	5

$$\text{If } s_i \neq t_j \text{ then } T[i,j] = \max(T[i-1,j], T[i,j-1])$$
$$\text{If } s_i = t_j \text{ then } T[i,j] = T[i-1,j-1] + 1$$

Example

1	2	3	4	5	6
t	r	a	i	n	
s	t	r	o	n	g

6	0	1	2	2	2	3
5	0	1	2	2	2	3
4	0	1	2	2	2	2
3	0	1	2	2	2	2
2	0	1	1	1	1	1
1	0	0	0	0	0	0
0	0	0	0	0	0	0
	0	1	2	3	4	5

Bactracking

If $s_i \neq t_j$ then $T[i,j] = \max(T[i-1,j], T[i,j-1])$

If $s_i = t_j$ then $T[i,j] = T[i-1,j-1] + 1$

1	2	3	4	5	6
t	r	a	i	n	
s	t	r	o	n	g

LCS: trn

6	0	1	2	2	2	3
5	0	1	2	2	2	3
4	0	1	2	2	2	2
3	0	1	2	2	2	2
2	0	1	1	1	1	1
1	0	0	0	0	0	0
0	0	0	0	0	0	0
	0	1	2	3	4	5

Recurrence

We can choose in which order we fill out the table:

```
for i=1 to m
  for j=1 to n
```

6	0	1	2	2	2	3
5	0	1	2	2	2	3
4	0	1	2	2	2	2
3	0	1	2	2	2	2
2	0	1	1	1	1	1
1	0	0	0	0	0	0
0	0	0	0	0	0	0
	0	1	2	3	4	5

or

```
for j=1 to n
  for i=1 to m
```

6	0	1	2	2	2	3
5	0	1	2	2	2	3
4	0	1	2	2	2	2
3	0	1	2	2	2	2
2	0	1	1	1	1	1
1	0	0	0	0	0	0
0	0	0	0	0	0	0
	0	1	2	3	4	5

Recurrence

We can choose in which order we fill out the table:

6	0	1	2	2	2	3
5	0	1	2	2	2	3
4	0	1	2	2	2	2
3	0	1	2	2	2	2
2	0	1	1	1	1	1
1	0	0	0	0	0	0
0	0	0	0	0	0	0
	0	1	2	3	4	5

Recurrence

Running time: $O(mn)$

Memory/Space: $O(mn)$

Questions?

Distances Between Strings

In numerous applications, we need to measure the distance between strings.

- Genomics: are two DNA sequences close to each other?
- Spell checking: find a word closest to a misspelled word
- Document search
- …

Distances Between Strings

- Hamming distance

The distance between two strings $s_1 \ldots s_m$ and $t_1 \ldots t_m$ equals the number of positions they differ at

$$d_H(s,t) = \left|\{j: s_j \neq t_j\}\right|$$

Usually, this is not a great choice.

1	2	3	4	5	6	7
a	d	r	e	s	s	_
a	d	d	r	e	s	s

$$d_H(adress\_, address) = 4$$

Distances Between Strings

➢ Insertion-Deletion Edit Distance

Consider the following process.

- start with s
- perform a number of steps
- at each step with either insert one character or delete one character from the current string
- finally, we obtain t

The insertion-deletion edit distance is the minimal number of steps we need to obtain t from s.

Distances Between Strings

- Insertion-Deletion Edit Distance

adress ⇒ address (1 insertion: $\text{dist} = 1$)

tommorow ⇒ tommorrow ⇒ tomorrow (1 deletion, 1 insertion: $\text{dist} = 2$)

Supstitution ⇒ Sustitution ⇒ Substitution (1 deletion, 1 insertion: $\text{dist} = 2$)

Distances Between Strings

Consider

- characters in s that have not been deleted in the process
- characters in t that have not been inserted in the process

These are the same characters.

We can match their occurrences in s and t. They form an LCS!

Denote their number by k. Then

- we deleted $m - k$ characters
- we inserted $n - k$ characters

The total number of operations is $(m + n) - 2k$

Distances Between Strings

The maximum possible value of k is $LCS(s,t)$.

Thus, the insertion-deletion edit distance is

$$d_{ins.del.edit}(s,t) = (m+n) - 2LCS(s,t).$$

DP for the Insertion-Deletion Edit Distance

- $T[0, j] = j$
- $T[i, 0] = i$

If $s_i \neq t_j$ then

$$T[i, j] = \min(T[i-1, j], T[i, j-1]) + 1$$

If $s_i = t_j$ then

$$T[i, j] = T[i-1, j-1]$$

Edit Distance

➢ (standard) Edit Distance. Three operations:

- Insertion
- Deletion
- Substitution

If $s_i \neq t_j$ then

$$T[i,j] = \min(T[i-1,j], T[i,j-1], T[i-1,j-1]) + 1$$

If $s_i = t_j$ then

$$T[i,j] = T[i-1,j-1]$$

Edit Distance

- Insertion cost: c_{ins}
- Deletion cost: c_{del}
- Substitution cost: c_{sub}

$$T[0,j] = c_{ins} \cdot j$$

$$T[i,0] = c_{del} \cdot i$$

If $s_i \neq t_j$ then

$$T[i,j] = \min(T[i-1,j] + c_{del}, T[i,j-1] + c_{ins}, T[i-1,j-1] + c_{sub})$$

If $s_i = t_j$ then

$$T[i,j] = T[i-1,j-1]$$

Distances Between Strings

Questions?

Bellman-Ford Algorithm

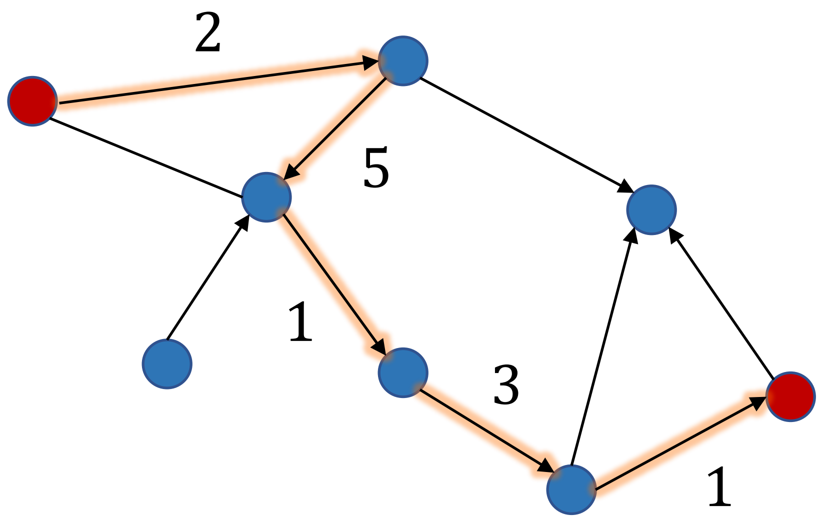
Consider a directed or undirected graph $G = (V, E)$

Assume that every edge e has length $c(e)$.

The length of a path P:

$u_1 \to u_2 \to \cdots \to u_k$ is

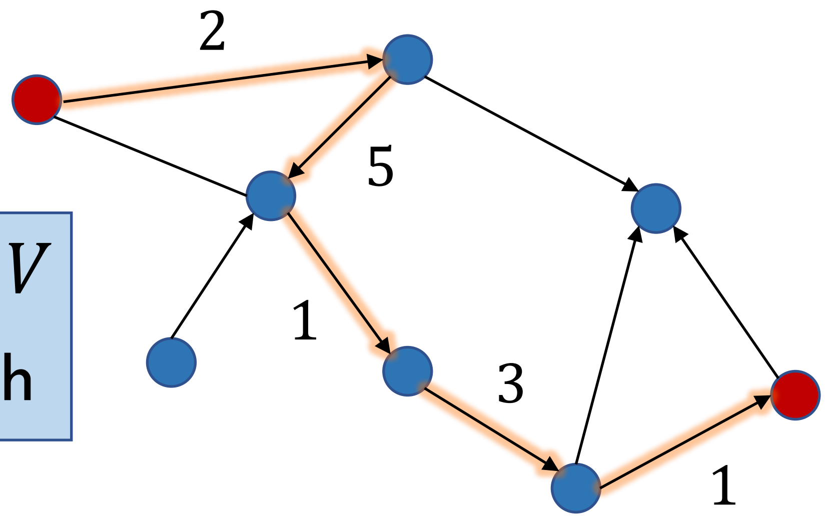
$$\text{length}(P) = \sum_{i=1}^{k-1} c(u_i, u_{i+1})$$



$2 + 5 + 1 + 3 + 1 = 12$

Bellman-Ford Algorithm

The distance between s and t equals the length of the shortest path from s to t.



Given: $G(V, E)$, lengths c, and vertex/source $s \in V$

Find the distance from s to all vertices in the graph

For now, assume that $c(u, v) \geq 0$.

DP for Shortest Path

- Subproblem (u, k)

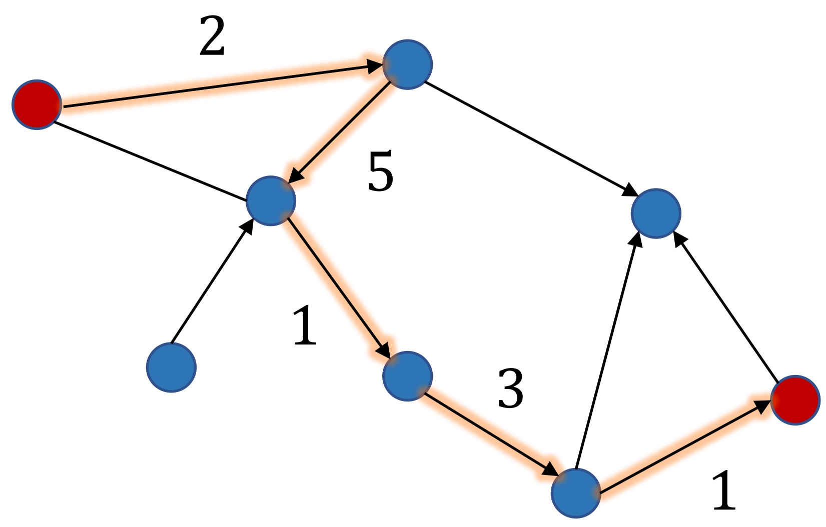
Find the shortest path from s to u with at most k edges.

- DP Table

Entries $T[u, k]$ where

- $u \in S$
- $k \in \{0, \dots, n-1\}$

Note that every simple path in G has at most $n-1$ edges.

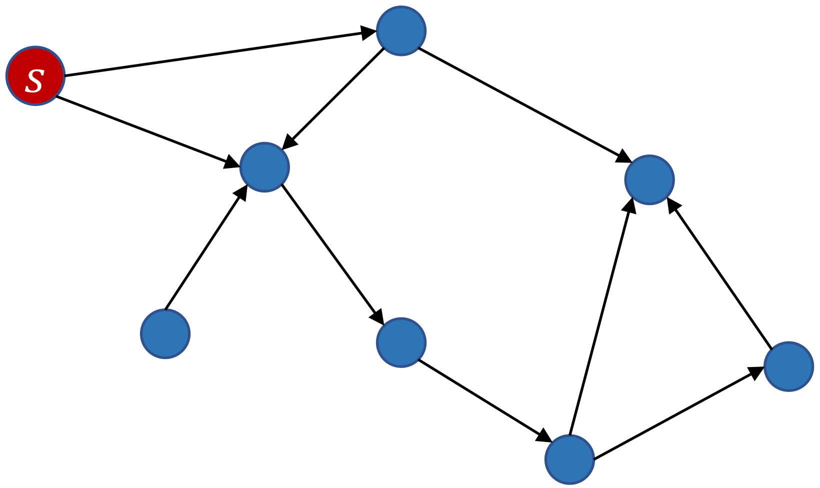


DP for Shortest Path

- Initialization

$T[s,*] = 0$

$T[u,0] = +\infty$ for $u \neq s$

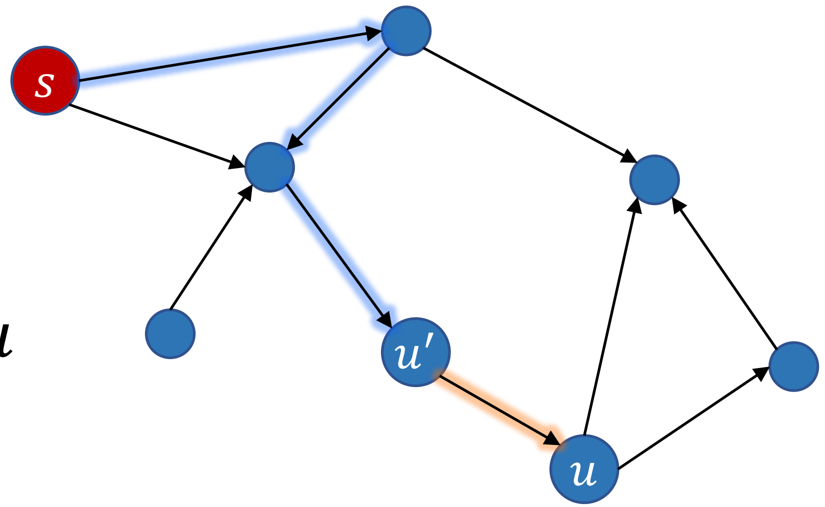


DP for Shortest Path

➢ Recurrence for $T[u, k]$

Consider the shortest path from s to u with $\mathrm{k}' \leq k$ edges.

$$P: s \equiv v_1 \to v_2 \to \cdots \to v_{k'} \to v_{k'+1} \equiv u$$



Path P consists of path P' from s to $u' = v_{k'}$ and edge (u', u).

- P' has at most $k' - 1 \leq k - 1$ edges.

Thus,

$$\mathrm{length}(P) = \mathrm{length}(P') + c(u', u) = T[u', k-1] + c(u', u)$$

DP for Shortest Path

Recurrence for $T[u, k]$

$$T[u,k] = \min_{(u',u)\in E}\big(T[u',k-1] + c(u',u)\big)$$

$T[u, 0] = +\infty$ for all $u \neq s$
$T[s,*] = 0$

for $k = 1$ to $n - 1$

  for $u \in V \setminus \{s\}$

    $$T[u,k] = \min_{(u',u)\in E}\big(T[u',k-1] + c(u',u)\big)$$

return $\{T[u, n-1]: u \in V\}$

DP for Shortest Path

$T[u, 0] = +\infty$ for all $u \neq s$
$T[s,*] = 0$

for $k = 1$ to $n - 1$

  for $u \in V \setminus \{s\}$

    $T[u, k] = \min_{(u',u)\in E}\left(T[u', k - 1] + c(u', u)\right)$

return $\{T[u, n - 1]: u \in V\}$

Running time: $O\left((n - 1) \times \sum_{u\in V} \deg_{in} u\right) = O(mn)$

Memory: $O\left(n^2\right) \leftarrow$ can we improve this?

DP for Shortest Path

we can store only values of $T[*,k]$ and $T[*,k-1]$

$T[u,0] = +\infty$ for all $u \neq s$
$T[s,*] = 0$

for $k = 1$ to $n-1$

for $u \in V \setminus \{s\}$

$$T[u,k] = \min_{(u',u)\in E}\left(T[u',k-1] + c(u',u)\right)$$

return $\{T[u,n-1]: u \in V\}$

Running time: $O\left((n-1) \times \sum_{u\in V} \deg_{in} u\right) = O(mn)$

Memory: $O\left(n^2\right) \leftarrow O(n)$

Ignore the second index?

$T[u] = +\infty$ for all $u \neq s$
$T[s] = 0$

for $k = 1$ to $n - 1$

  for $u \in V \setminus \{s\}$

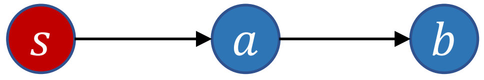
    $T[u] = \min_{(u',u)\in E} \big(T[u'] + c(u',u)\big)$

return $\{T[u] : u \in V\}$

Is this algorithm equivalent to the previous?

$$T[u] \overset{?}{=} T[u,k] \quad \text{after iteration } k$$

The second index?



$T[u] = +\infty$ for all $u \neq s$
$T[s] = 0$

for $k = 1$ to $n - 1$

  for $u \in V \setminus \{s\}$

    $$T[u] = \min_{(u',u)\in E} \left(T[u'] + c(u', u)\right)$$

return $\{T[u] : u \in V\}$

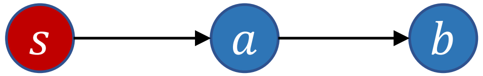
$k \setminus u$	s	a	b
0	0	∞	∞
1	0	1	∞
2	0	1	2

iter.	s	a	b
0	0	∞	∞
1	0	1	2
2	0	1	2

Is this algorithm equivalent to the previous?

$$T[u] \stackrel{?}{=} T[u, k] \quad \text{after iteration } k$$

The second index?



$k \setminus u$	s	a	b
0	0	∞	∞
1	0	1	∞
2	0	1	2

iter.	s	a	b
0	0	∞	∞
1	0	1	2
2	0	1	2

The execution of the original algorithm

for $u \in V \setminus \{s\}$

$$T[u,k] = \min_{(u',u)\in E}\big(T[u',k-1] + c(u',u)\big)$$

doesn't depend on the order in which we go over all $u \in V$.

The execution of the new algorithm does.

New algorithm: analysis

Claim

I. $T[u] \leq T[u,k]$ after iteration k

II. There is a path from s to u of length at most $T[u]$

I. Proof by induction. Assume that I holds till the current assignment. We let

$$T[u] = \min_{(u',u)\in E}\left(T[u'] + c(u',u)\right)$$

$\leq$ by the induction hypothesis

But

$$T[u,k] = \min_{(u',u)\in E}\left(T[u',k-1] + c(u',u)\right)$$

Thus, $T[u] \leq T[u,k]$ as required.

New algorithm: analysis

Claim

I. $T[u] \leq T[u,k]$ after iteration k

II. There is a path from s to u of length at most $T[u]$

II. Proof by induction. Assume that II holds till the current assignment and

$$T[u] = T[u'] + c(u', u)$$

By induction hypothesis, there is a path P' from s to u' of length $T[u']$.

Add edge (u', u) to P'. We obtain the desired path of length $T[u]$.

New algorithm: analysis

Claim

I. $T[u] \leq T[u,k]$ after iteration k

II. There is a path from s to u of length at most $T[u]$

Conclusion. After iteration $n-1$.

$$T[u] \leq T[u,n-1] = d(s,u)$$
$$T[u] = \text{length}(P_{s\to u}) \geq d(s,u)$$

Thus, $T[u] = d(s,u)$.

Number of iterations?

We proved that the algorithm finds the correct answer.

Before it stops, it performs $n - 1$ iterations (for $k = 1$ to $n - 1$).

Do we need all $k - 1$ iterations?

Consider a path graph $G = P = u_1 \to u_2 \to \cdots \to u_n$ and $s = u_1$

iter.	$s = u_1$	u_2	u_3	u_4	…	…	…	…	…	u_n
0	0	∞	∞	∞	…	…	…	…	…	∞
1	0	1	2	3	…	…	…	…	…	∞
2	0	1	2	3	…	…	…	…	…	∞

Negative edge lengths/costs

The algorithm works correctly if some edges have negative lengths as long as there are negative cycles.

We used positivity of $c(u, v)$ only to claim that the shortest path is a simple path and thus

$$d(s, u) = T[u, n - 1]$$

Advantages/Disadvantages

- Slower than Dijkstra's algorithm

$$O(mn) \quad vs. \quad O(m \log n)$$

+ Can handle negative edges costs
+ Easy to implement. Don’t need any more advanced datastructures.
+ Easy to parallelize.
+ The running time is reasonable if the graph is moderately large.