Algorithms and Data Structures

Lecture 11

Dynamic Programming

Fabian Kuhn
Algorithms and Complexity

UNI FREIBURG

Dynamic Programming (DP)

- Important algorithm design technique!
- Simple, but very often a very effective idea

- Many problems that naively require exponential time can be solved in polynomial time by using dynamic programming.
  - This in particular holds for optimization problems (min / max)

DP $\approx$ careful / optimized brute force solution

DP $\approx$ recursion + reuse of partial solutions

DP: History

- Where does the name come from?
- DP was developed by Richard E. Bellman in the 1940s and 1950s. In his autobiography, he writes:

  "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. … The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. … His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. … Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. … It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. … Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. …"

Fibonacci Numbers

Definition of the Fibonacci numbers $F_0, F_1, F_2, \ldots$:

$$F_0 = 0, F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

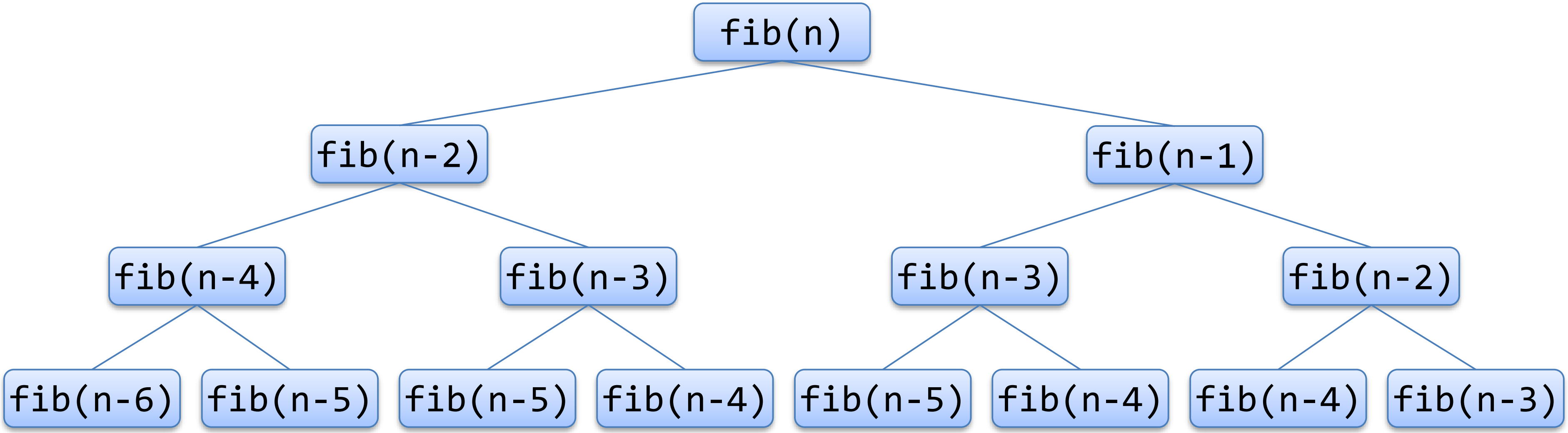
Goal: Compute F_n

- This can easily be done recursively…

```
def fib(n):
    if n < 2:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

Running Time of Recursive Algorithm

```
def fib(n):
    if n < 2:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    return f
```



- Recursion tree is a binary tree that is complete up to depth $n/2$.
- Running time : $\Omega\left(2^{n/2}\right)$
  - We repeatedly compute the same things!

Algorithm with Memoization

Memoization: One stores already computed values
(on a notepad = memo)

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n < 2:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

creates a new dictionary (a hash table)

First check,if we have already computed `fib(n)`.

Store the computed value for `fib(n)` in the hash table.

- Now, each value `fib(i)` is only computed once recursively
  - For every i we only go once through the blue part.
  - The recursion tree therefore has $\leq n$ inner nodes.
  - The running time is therefore $O(n)$.

DP: A bit more precisely …

DP $\approx$ Recursion + Memoization

Memoize: *Store* solutions for *subproblems*, reuse stored solutions if the same subproblem appears again.

- For the Fibonacci numbers, the subproblems are $F_0, F_1, F_2, \dots$

Running Time $=$ #subproblems $\cdot$ time per subproblem

Usually just the number of recursive calls per subproblem.

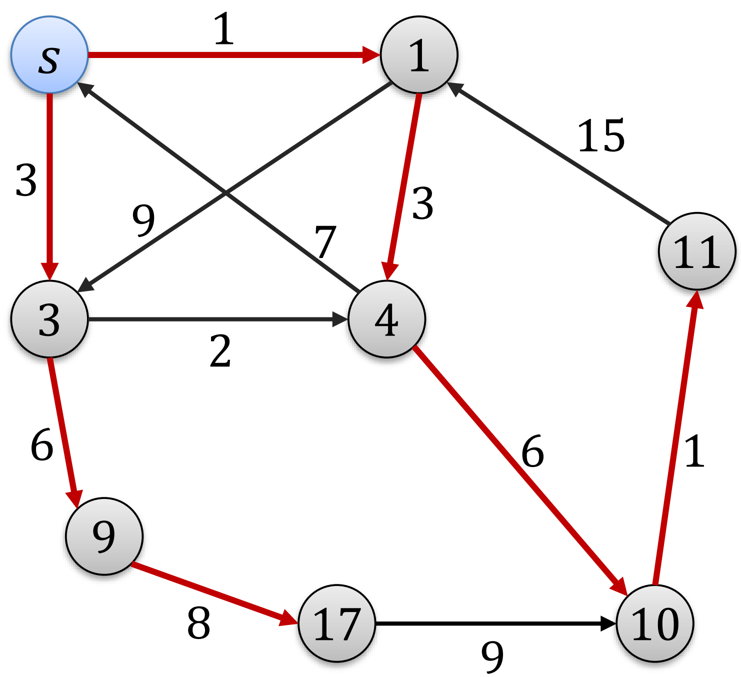
Fibonacci: Bottom-Up

```
def fib(n):
    fn = {}
    for k in [0,1, 2, …, n]:
        if k < 2:
            f = k
        else:
            f = fn[k-1] + fn[k-2]
        fn[k] = f
    return fn[n]
```

- Go through the subproblms in an order such that one has always already computed the subproblems that one needs.
  – In the case of the Fibonacci numbers, compute F_{i-2} and F_{i-1}, before computing F_i.

Shortest Paths with DP

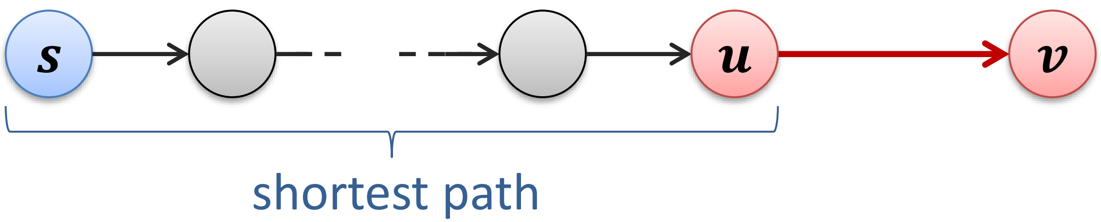
- **Given:** weighted, directed graph $G = (V, E, w)$
  - starting node $s \in V$
  - We denote the weight of an edge (u, v) as $w(u, v)$
  - Assumption: $\forall e \in E: w(e)$, no negative cycles
- **Goal:** Find shortest paths / distances from s to all nodes
  - Distance from s to v: $d_G(s, v)$ (length of a shortest path)



Shortest Paths : Recursive Formulation

Recursive characterization of $d_G(s,v)$?

- How does a shortest path from s to v look like?
- **Optimality of subpaths:**
  If $v \neq s$, then there is a node u, such that the shortest path consists of a shortest path from s to u and the edge (u,v).



$$\forall v \neq s \;:\; d_G(s,v) = \min_{u \in N_{\text{in}}(v)} d_G(s,u) + w(u,v)$$

- Can we use this to compute the values $d_G(s,v)$ recursively?

Shortest Paths : Recursive Formulation

Recursive characterization of $d_G(s,v)$?

$$d_G(s,v) = \min_{u \in N_{\text{in}}(v)} \{d_G(s,u) + w(u,v)\}, \qquad d_G(s,s) = 0$$

```
dist(v):
    d = ∞
    if v == s:
        d = 0
    else:
        for (u,v) in E:
            d = min(d, dist(u) + w(u,v))
    return d
```

Problem: cycles!

- With cycles we obtain an infinite recursion
  - Example: Cycle of length 2 (edges (u,v) and (v,u))
  - `dist(v)` calls `dist(u)`, `dist(u)` then again calls `dist(v)`, etc.

Shortest Paths in Acyclic Graphs

UNI FREIBURG

```
memo = {}
dist(v):
    if v in memo: return memo[v]
    d = ∞
    if v == s:
        d = 0
    else:
        for (u,v) in E:          (go through all incoming edges of v)
        d = min(d, dist(u) + w(u,v))
    memo[v] = d
    return d
```

Running time: $O(m)$

- Number of subproblems: n
- Time for subproblem $d_G(s, v)$: #incoming edges of v

Acyclic Graphs: Bottom-Up

Observation:

- Edge $(u, v) \implies d_G(s, u)$ must be computed before $d_G(s, v)$
- One can first compute a topological sort of the nodes.

Assumption:

- Sequence $v_1, v_2, \ldots, v_n$ is a topological sort of the nodes.

```
D = “array of length n”
for i in [1:n]:
    D[i] = ∞
    if v_i == s:
        D[i] = 0
    else:
        for (v_j, v_i) in E:        (incoming edges, top. sort ⟹ j < i)
        D[i] = min(D[i], D[j] + w(v_j, v_i))
```

Shortest Paths in General Graphs

Idea: Introduce additional subproblems to
avoid cyclic dependencies

Subproblems $d_G^{(k)}(s,v)$

- Length of shortest path consisting of at most k edges

Recursive Definition:

$$d_G^{(k)}(s,v) = \min\left\{d_G^{(k-1)}(s,v),\ \min_{(u,v)\in E}\left\{d_G^{(k-1)}(s,u) + w(u,v)\right\}\right\}$$

$$d_G^{(k)}(s,s) = 0, \qquad (\forall k \geq 0)$$

$$d_G^{(0)}(s,v) = \infty, \qquad (\forall v \neq s)$$

Shortest Paths in General Graphs

```
memo = {}
dist(k, v):
    if (k, v) in memo: return memo[(k, v)]
    d = ∞
    if s == v:
        d = 0
    elif k > 0:
        d = dist(k-1, v)
        for (u,v) in E:
                                    (go through all incoming edges of v)
            d = min(d, dist(k-1, u) + w(u,v))
    memo[(k, v)] = d
    return d

distance(v):
    return dist(n-1, v)
```

Shortest Paths with DP: Running Time

DP running time, typically:

$$\#\textbf{subproblems} \cdot \textbf{time per subproblem}$$

- Time per subproblem: recursive call costs 1 time unit
  - Because of memoization, every subproblem is only called once
  - Recursive cost is therefore captured by the first factor.
- Time per subproblem: typically #recursive possibilities

Shortest Paths:

- #subproblems: $O(n^2)$
- Time per subproblem: #incoming edges

Running Time: $O(m \cdot n)$

- Same running time as for Bellman-Ford
  - Algorithm essentially also corresponds to the Bellman-Ford algorithm.

Shortest Paths: Bottom-Up

- Usually, dynamic programs are writte bottom-up
  - It is often more efficient (no recursion, no hash table)
  - It is often a natural formulation of the algorithm.

- Bottom-Up DP Algorithm
  - Requires order in which the subproblems can be computed (topological sort of the dependency graph)
  - As we anyway have to make sure that there are no cyclic dependencies, this topological sort can usually be optained very easily.

- Order for the Shortest Paths problem
  - Sort $d_G^{(k)}(s, v)$ by k (increasingly)
  - For equal k-values, there are no dependencies

Shortest Paths: Bottom-Up

```
dist = “2-dimensional array”
for k in range(n):
    for v in V:
        d = ∞
        if v == s:
            d = 0
        elif k > 0:
            d = dist[k-1, v]
            for (u,v) in E:
                              (go through all incoming edges of v)
            d = min(d, dist[k-1, u] + w(u,v))
        dist[k, v] = d
```

5 Steps to a DP Solution

5 Steps	Analysis
1) Define subproblems	Count #subproblems
2) Guess (part of solution)	Count #possibilities
3) Set up recursion formula	Time per subproblem
4) Recursion + Memoization or set up bottom-up DP table	time = time per subproblem · #subproblems
5) Solve original problem	Possibly requires additional time

- Dynamic programming is a good approach if a problem can be solved recursively such that the number of possible different subproblems that one has to solve recursively is relatively small.

5 Steps to a DP Solution

5 Steps	Fibonacci Number F_n
1) Define subproblems	#subproblems $= n$
2) Guess (part of solution)	nothing to guess, #possibilities $= 1$
3) Set up recursion formula	Time per subproblem $= O(1)$
4) Recursion + Memoization or set up bottom-up DP table	Time $=$ time per subproblem $\cdot$ #subproblems $= O(1) \cdot n = O(n)$
5) Solve original problem	Lösung ist Teilproblem F_n, Zeit $O(1)$

5 Steps	Single Source Shortest Paths (Bellman-Ford)
1) Define subproblems	#subproblems $= n \cdot (n-1)$ (alle $d_G^{(k)}(s,v)$)
2) Guess (part of solution)	$d_G^{(k)}(s,v)$: edge to v, #possibilities: $1 +$ in-degree of v
3) Set up recursion formula	Time per subproblem $= \Theta\big(1 + \text{in\_degree}(v)\big)$
4) Recursion + Memoization or set up bottom-up DP table	Time $= \sum_{\text{subproblems}} \text{time per subproblem}$ $= \sum_{v \in V} \Theta\big(1 + \text{in\_degree}(v)\big) = \Theta(\|V\| \cdot \|E\|)$
5) Solve original problem	All $d_G^{(n-1)}(s,v)$, time $O(\|V\|)$

Computing the Solution

Recursive Computation of the Optimization Function

- All possibilities are tested (recursively)
- The best one (min/max) is chosen

Computing the Solution

- The recursive call for the optimization function only returns the optimal function value (e.g., length of a shortest path).
- To obtain the recursively computed solution, one has to remember, which of the possibilities in each step gives the optimal value.
- If doing DP with a hash table, this information is also stored in the hash table.
- Bottom-up: In each cell of the table, one not only stores the value, but also how the value was obtained.

Computing Solution: Parent Pointers

General DP

```
memo = {}
parent = {}
DP(x1, x2, …, xk):
    if (x1, x2, …, xk) in memo:
        return memo[(x1, x2, …, xk)]
    if (x1, x2, …, xk) in base
        value = …
    else:
        value = min/max of the value of DP(x1, x2, …, xk)
                over predecessor node (y1, y2, …, yk) in
                the dependency graph
    memo[(x1, x2, …, xk)] = value
    parent[(x1, x2, …, xk)] = (y1, y2, …, yk)-tuple that
                              achieved the min/max
    return value
```

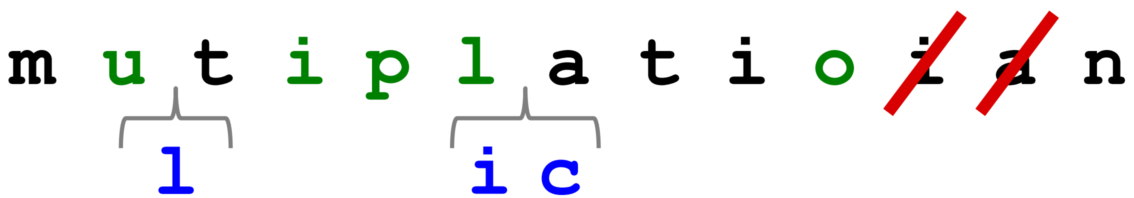
Edit Distance

- For two strings A and B, compute

  Edit Distance $D(A, B)$ (# edit op., to transform A into B)

  and also a minimal sequence of edit operations to transform A into B.

- **Example:** mathematician → multiplication:



Edit Distance

Given: Two strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$

Goal: Determine the minimum number $D(A, B)$ of edit operations required to transform A into B

Edit operations:

a) **Replace** a character from string A by a character from B

b) **Delete** a character from string A

c) **Insert** a character from string B into A

```
m a – t h e m – – a t i c i a n
m u l t i p l i c a t i o – – n
```

Edit Distance : Cost Model

- Cost for **replacing** character a by b: $\boldsymbol{c(a,b) \geq 0}$

- Capture insert, delete by allowing $a = \varepsilon$ or $b = \varepsilon$:
  - Cost for **deleting** character a: $\boldsymbol{c(a,\varepsilon)}$
  - Cost for **inserting** character b: $\boldsymbol{c(\varepsilon,b)}$

- **Triangle inequality**:

$$c(a,c) \leq c(a,b) + c(b,c)$$

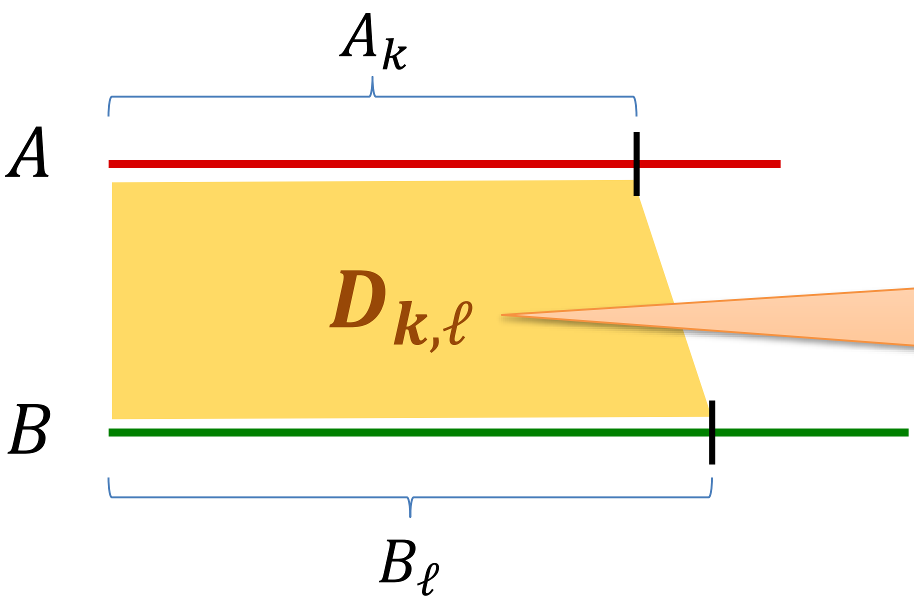
→ each character is changed at most once!

- **Unit cost model**: $c(a,b) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \end{cases}$

Edit Distance: Subproblems

Define $A_k := a_1 \dots a_k$, $B_\ell := b_1 \dots b_\ell$

Subproblems: $\boldsymbol{D_{k,\ell} := D(A_k, B_\ell)}$



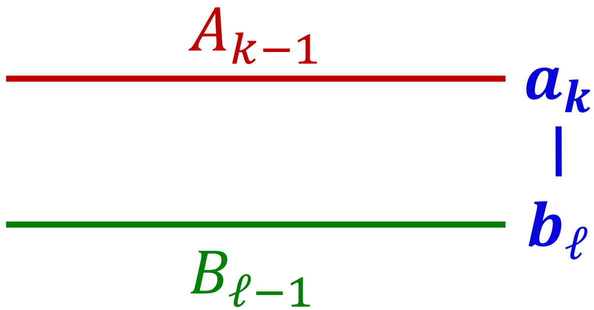
Edit distance between prefix A_k of A and prefix B_ℓ of B

Computing the Edit Distance

Three ways to end optimal "alignment" between A_k and B_ℓ:

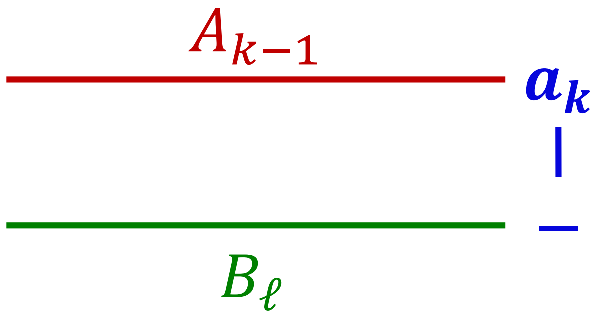
1. a_k is replaced by b_ℓ:

   $$D_{k,\ell} = D_{k-1,\ell-1} + c(a_k, b_\ell)$$



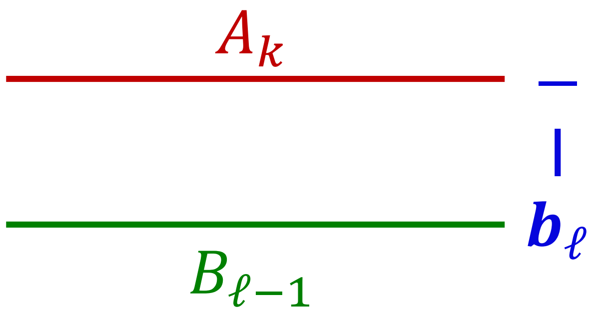
2. a_k is deleted:

   $$D_{k,\ell} = D_{k-1,\ell} + c(a_k, \varepsilon)$$



3. b_ℓ is inserted:

   $$D_{k,\ell} = D_{k,\ell-1} + c(\varepsilon, b_\ell)$$

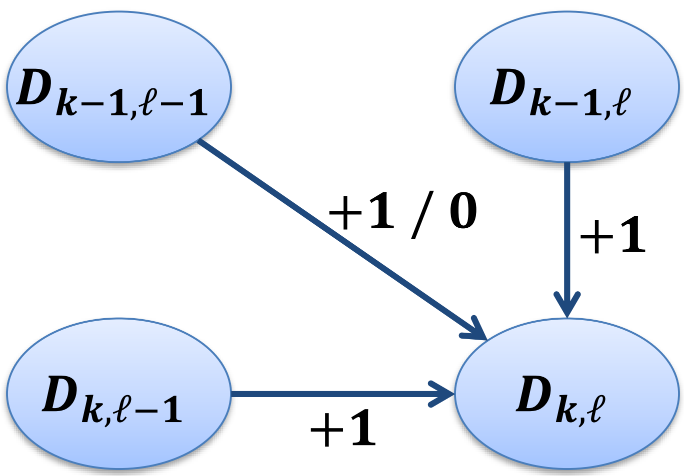


Computing the Edit Distance

- Recurrence relation (for $k, \ell \geq 1$)

$$D_{k,\ell} = \min \begin{Bmatrix} D_{k-1,\ell-1} & + c(a_k, b_\ell) \\ D_{k-1,\ell} & + c(a_k, \varepsilon) \\ D_{k,\ell-1} & + c(\varepsilon, b_\ell) \end{Bmatrix} = \min \underbrace{\begin{Bmatrix} D_{k-1,\ell-1} & + 1 \;/\; 0 \\ D_{k-1,\ell} & + 1 \\ D_{k,\ell-1} & + 1 \end{Bmatrix}}_{\textbf{unit cost model}}$$

- Need to compute $D_{i,j}$ for all $0 \leq i \leq k$, $0 \leq j \leq \ell$:



Recursion Equation of Edit Distance

Base cases:

$$D_{0,0} = D(\varepsilon, \varepsilon) = 0$$

$$D_{0,j} = D(\varepsilon, B_j) = D_{0,j-1} + c(\varepsilon, b_j) = j$$

$$D_{i,0} = D(A_i, \varepsilon) = D_{i-1,0} + c(a_i, \varepsilon) = i$$

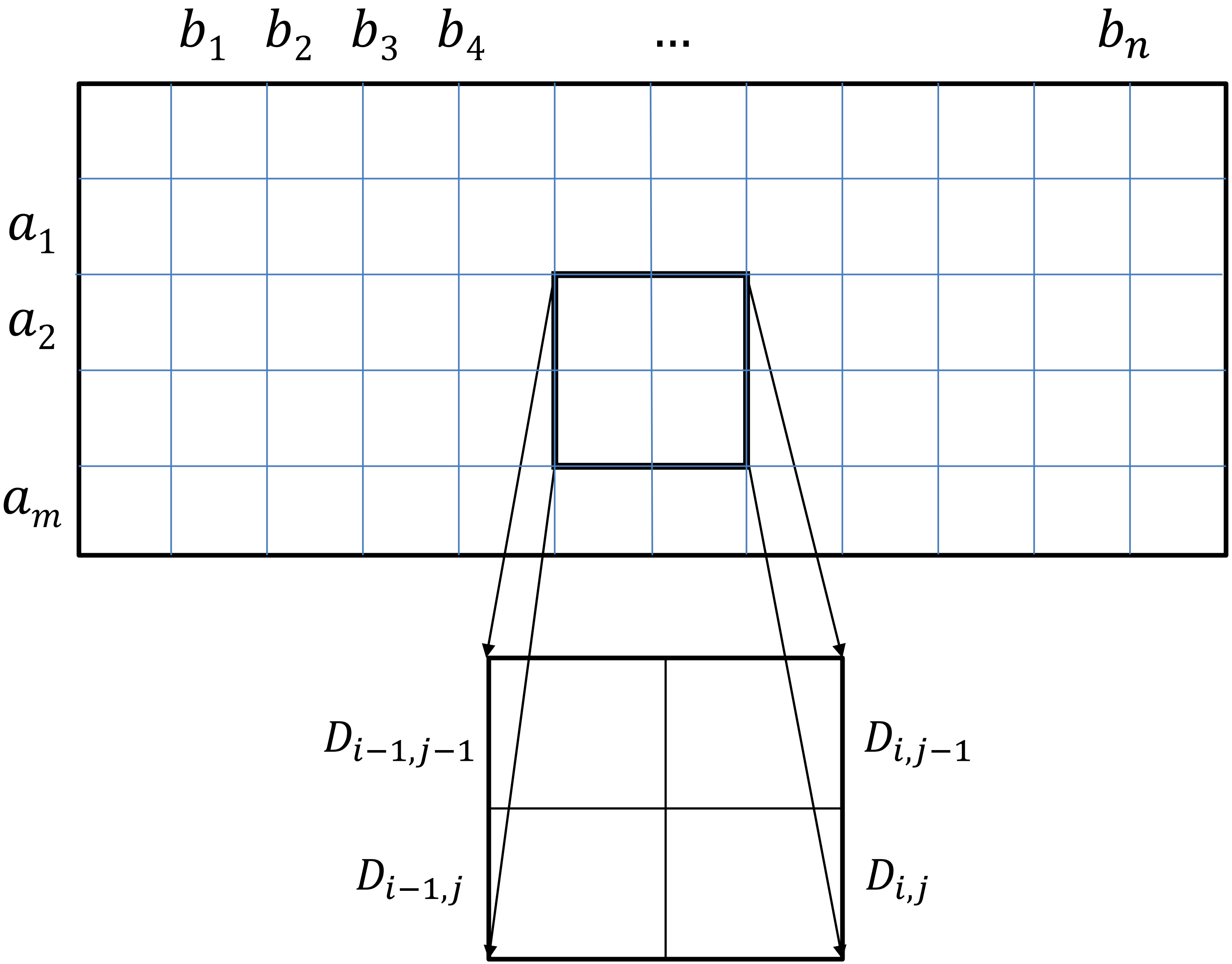
unit cost model

Recurrence relation:

$$D_{i,j} = \min \begin{Bmatrix} D_{k-1,\ell-1} & + c(a_k, b_\ell) \\ D_{k-1,\ell} & + c(a_k, \varepsilon) \\ D_{k,\ell-1} & + c(\varepsilon, b_\ell) \end{Bmatrix} = \min \begin{Bmatrix} D_{k-1,\ell-1} & + 1 / 0 \\ D_{k-1,\ell} & + 1 \\ D_{k,\ell-1} & + 1 \end{Bmatrix}$$

unit cost model

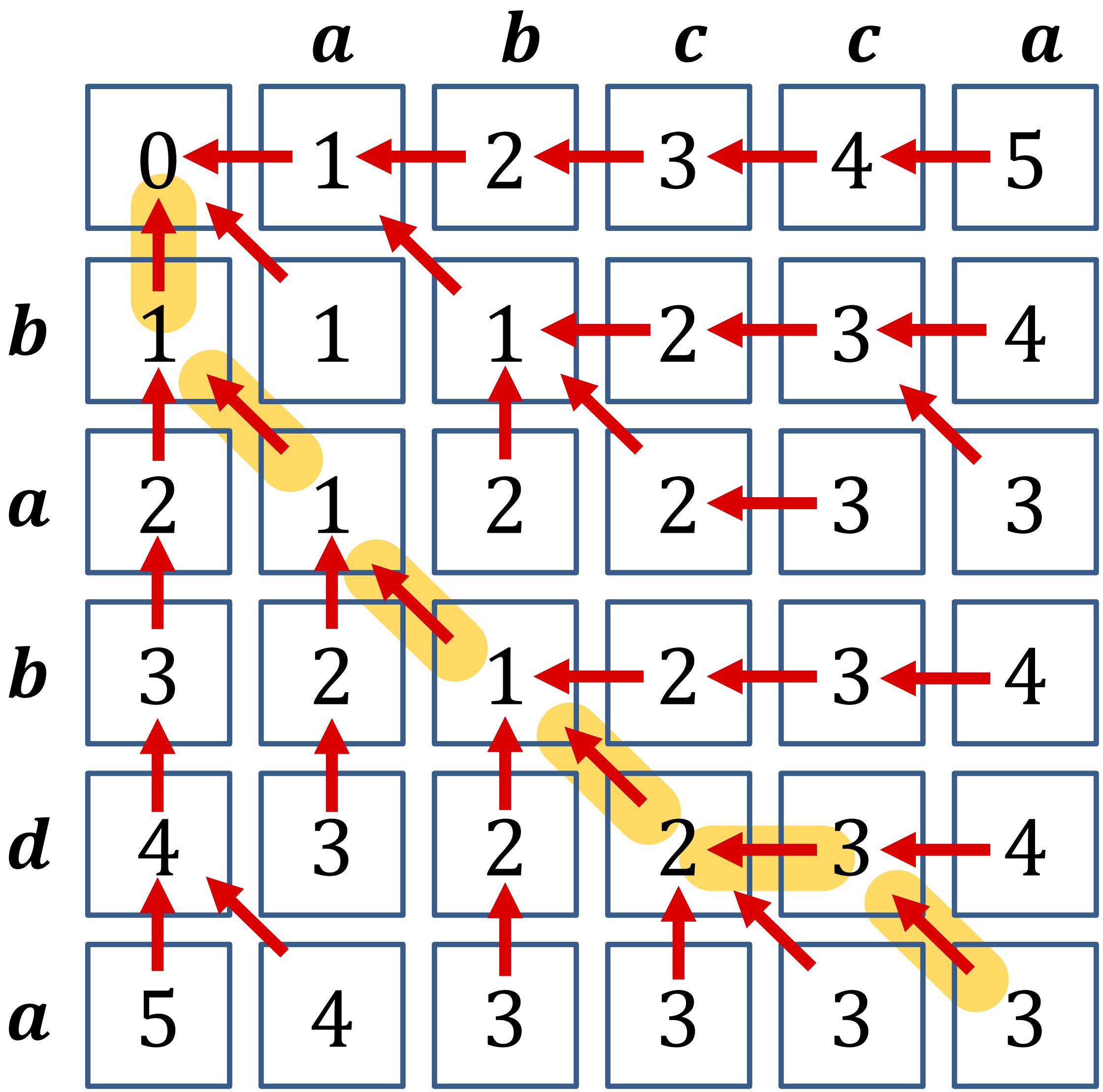
Order of the Subproblems



Example

		a	b	c	c	a
b						
a						
b						
d						
a						

Edit Operations



b a b d - a
- a b c c a

Edit Distance – Summary

- **Running Time:**
  - Edit distance between two strings of lengths m and n can be computed in $O(m \cdot n)$ time.
- **Obtain the edit operations:**
  - for each cell, store which rule(s) apply to fill the cell
  - track path backwards from cell (m, n)
- **Unit cost model:**
  - interesting special case, each edit operation costs 1
- **Optimization:**
  - If the edit distance is small, we do not need to fill out the whole table.
  - If the edit distance is $\leq \delta$, only entries at distance at most δ from the main diagonal of the table are really relevant.
  - For two strings of length n, we then only have to fill out $O(\delta \cdot n)$ entries.
  - With this idea, one can compute the edit distance in time $O\big(n \cdot D(A, B)\big)$.