Minimum Edit Distance

$S = {}_sS'$

$T = {}_tT'$

Optimal Substructure

- The min edit distance for (S,T) can be computed from the min edit distance of (S',T') (S,T') (S',T)

  substitution insertion deletion

Example

- If $s=t$ the min edit distance of (S,T) is the same as (S',T')
  - If we can transform S' to T' in K edits then we can transform S into T in the same K edits

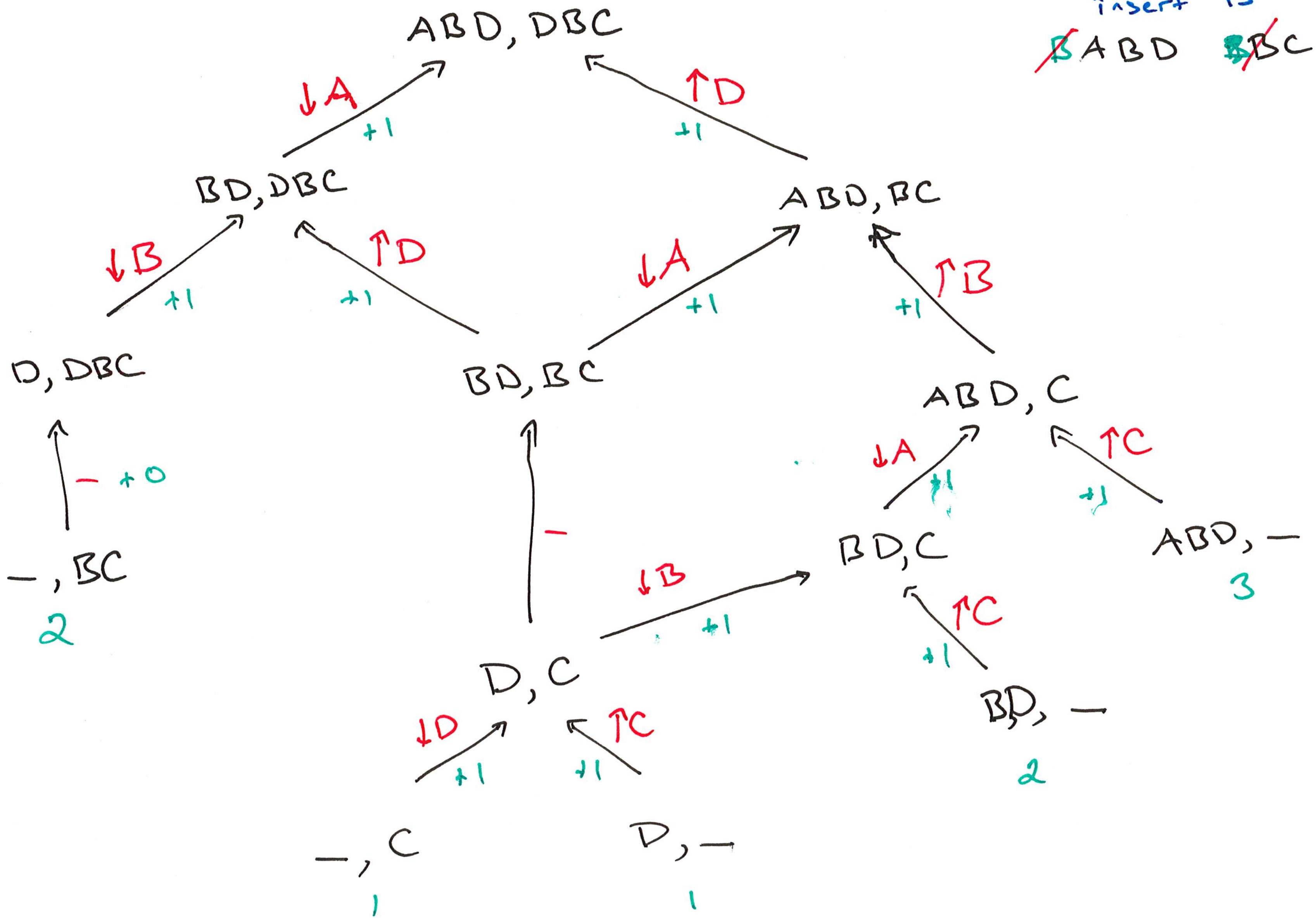
- To show k is minimal, i.e. that we cannot transform S into T in fewer than k edits.
  - Suppose, for contradiction, that we can transform S into T in $k' < k$ edits
  - we can assume that none of the k' edits affect the first characters. of S and T
  - therefore the same k' edits can be used to transform S' into T'

Overlapping Subproblems

only insertion, deletion

insert B

~~B~~ABD ~~B~~BC



- For two strings S and T

  $m = |S|$ $n = |T|$

- Define $D[i,j]$ as the edit distance between $S[1 \ldots i]$ and $T[1 \ldots j]$

- The edit distance between S and T is $D[m,n]$

- Initialize $D[i,0] = i$ and $D[0,j] = j$

- For all other values

$$D[i,j] = \min \begin{cases} D[i-1,j] + 1 \\ D[i,j-1] + 1 \\ D[i-1,j-1] + \begin{cases} 0 & \text{if } S_i = T_j \\ 1 & \text{otherwise} \end{cases} \end{cases}$$

N	6	6	5	4	3	3	2	$\boxed{3}$
E	5	5	4	3	2	2	3	4
T	4	4	3	2	1	2	3	4
T	3	3	2	1	2	3	4	5
I	2	2	1	2	3	4	5	6
K	1	1	2	3	4	5	6	7
#	0	1	2	3	4	5	6	7
	#	S	I	T	T	I	N	G

MED(S, T)

```
let D be a m × n array
for i = 0 to m
        D[i, 0] = i
for j = 0 to n
        D[0, j] = j
for j = 0 to n
        for i = 0 to m
                if S[i] == T[j]
                        D[i, j] = D[i − 1, j − 1]
                else
                        d = D[i − 1, j]
                        e = D[i, j − 1]
                        f = D[i − 1, j − 1]
                        D[i, j] = 1 + min(d, e, f)
return D[m, n]
```

$+1$ $+1$

$\Theta(m)$

$\Theta(n)$

$\cancel{\Theta(m,n)}$

$\Theta(mn)$

Time: $\Theta(mn)$

Space: $\cancel{\Theta(m,n)}$

$\Theta(mn)$

Dynamic Programming Summary

- ~~Op~~ Optimal Substructure: an optimal solution to a problem can be constructed from optimal solutions to subproblems.

- Overlapping Subproblems: subproblem appear more than once during computation

- Two main approaches
  - memoization → top-down
  - tabulation → bottom-up

Greedy Algorithms

- make an irrevocable choice
- recurse on a smaller subproblem
- combine the sub-result with our choice

In order to use a greedy algorithm the problem must have 2 properties.

1. Optimal substructure
2. Greedy choice property
   - there must be a way to select a next step that generates an overall optimal solution.

Greedy Efficiency

- Since we operate sequentially we get a natural lower bound of $\Omega(n)$

- Making a greedy choice might require a non-trivial amount of work.

    - we might need to maintain a dynamic data structure $\omega(1)$

    - we might need to preprocess the data in $\omega(n)$

- generally we get super linear algorithms overall.