# Dynamic Programming

We have seen that recursive search usually has exponential time $\Theta(b^n)$ where $n$ is the depth of recursion and $b$ is some branching factor, often $2$ or more.

This is because a lot of subinstances are generated.

But! How many subinstances are unique?

For the longest path problem, each node creates a single unique subinstance: $\Theta(|V|)$

For minimum edit distance, each pair of indices is a unique subinstance: $\Theta(x.\,\mathrm{length} \times y.\,\mathrm{length})$.

If recursive backtracking search repeatedly solves the same instances over and over, it is far less efficient than necessary.

# Top-down dynamic programming

Let's add a table $C$ to the longest path's recursive search solution.

Recall $\mathrm{dpl}(u,t) = \max_{p \mid u \xrightarrow{p} t} \mathrm{distance}(p)$

```
var C : V → Int ∪ {nil, −∞}
for u : V do C(u) := nil end for

procedure distanceOfLongestPath(u : V, t : V) : Int
maintains C(v) = dlp(v, t) or C(v) = nil, for each v
postcondition result = C(u) = dpl(u, t)
        and ∀v · C₀(v) ≠ nil ⇒ C(v) ≠ nil
    if C(u) = nil ∧ u = t then
        C(u) := 0
    elsif C(u) = nil then
        var best := −∞
        for v such that u → v do
            val cost := w(u, v)+distanceOfLongestPath(v)
            if cost > best then best := cost end if
        end for
        C(u) := best
    end if
    return C(u)
end distanceOfLongestPath

distanceOfLongestPath(s)
```

Now the cost of a longest cost path from $s$ to $t$ is in $C(s)$.

Time: $\Theta(|V|)$

Similarly, for the minimum edit distance. Define $\mathrm{med}(i,j)$ to be the minimal cost to transform $x[0,..i]$ to $y[0,..j]$.

var $C : \{0,..,x.\mathrm{length}\} \times \{0,..y.\mathrm{length}\} \to \mathrm{Int} \cup \{\mathrm{nil}\}$
for $(i,j)\ :\ \{0,..,x.\mathrm{length}\} \times \{0,..y.\mathrm{length}\}$ do
$C(i,j) := \mathrm{nil}$ end for

procedure $minEditDistance(i,j) : \mathrm{Int}$
precondition $0 \le i \le x.\mathrm{length}\ \wedge\ 0 \le j \le y.\mathrm{length}$
maintains $\forall i', j' \cdot C(i',j') = \mathrm{med}(i',j')$ or $C(i',j') = \mathrm{nil}$
postcondition $result = C(i,j) = \mathrm{med}(i,j)$
    and $\forall i', j' \cdot C_0(i',j') \neq \mathrm{nil} \Rightarrow C(i',j') \neq \mathrm{nil}$
  if $C(i,j) = \mathrm{nil} \wedge i = 0$ then $C(i,j) := j$
  elsif $C(i,j) = \mathrm{nil} \wedge j = 0$ then $C(i,j) := i$
  elsif $C(i,j) = \mathrm{nil} \wedge x(i-1) = y(j-1)$ then
    $C(i,j) := minEditDistance(i-1, j-1)$
  elsif $C(i,j) = \mathrm{nil}$ then
    val $rCost := 1 +\ minEditDistance(i-1, j-1)$
    val $dCost := 1 + minEditDistance(i-1, j)$
    val $iCost := 1 + minEditDistance(i, j-1)$
    $C(i,j) := \min(rCost, dCost, iCost)$
  end if
  return $C(i,j)$
end $minEditDistance$

$minEditDistance(x.\mathrm{length}, y.\mathrm{length})$

For the example, we get the following cost table. The arrows show the data flow through the table.

| | | $y =$ | R | | E | | N | | D | |
|---|---|---|---|---|---|---|---|---|---|---|
| $x$ | $i \searrow j$ | 0 | | 1 | | 2 | | 3 | | 4 |
| $=$ | 0 | 0 | | 1 | | 2 | | 3 | | nil |
| F | | | $\searrow$ | $\downarrow$ | $\searrow$ | $\downarrow$ | $\searrow$ | $\downarrow$ | | |
| | 1 | 1 | $\rightarrow$ | 1 | $\rightarrow$ | 2 | $\rightarrow$ | 3 | | nil |
| R | | | $\searrow$ | | $\searrow$ | $\downarrow$ | $\searrow$ | $\downarrow$ | | |
| | 2 | nil | | 1 | $\rightarrow$ | 2 | $\rightarrow$ | 3 | | nil |
| E | | | | | $\searrow$ | | $\searrow$ | $\downarrow$ | | |
| | 3 | nil | | nil | | 1 | $\rightarrow$ | 2 | | nil |
| D | | | | | | | | | $\searrow$ | |
| | 4 | nil | | nil | | nil | | nil | | 2 |

Abstracting, we get the following algorithm pattern.

Let $Z$ be a set that includes the root instance $R$ and all its descendent instances.

Let $opt(J)$ be the cost of an optimal solution for subinstance $J$.

var $C : Z \to Cost \cup \{\text{nil}\}$
for $I : Z$ do $C(I) := \text{nil}$ end for

procedure $topDownDynamic(I) : Cost$
maintains $\forall J \cdot C(J) \neq \text{nil} \Rightarrow C(J) = opt(J)$
postcondition $result = C(I) = opt(I)$
    and $\forall J \cdot C_0(J) \neq \text{nil} \Rightarrow C(J) \neq \text{nil}$
  if $C(I) = \text{nil} \wedge I$ *is a leaf instance* then
    $C(I) := opt(I)$
  elseif $C(I) = \text{nil}$ then
    let $K$ be *the number of subinstances of* $I$
    for $k \leftarrow \{0, ..K\}$ do
      $topDownDynamic(subinstance_k)$
    end for
    *compute* $C(I)$ *from the entries in* $C$ *corresponding to* $I$*'s direct subinstances*
  end if
  return $C(I)$
end $topDownDynamic$

Now $topDownDynamic(R)$ computes the cost of $R$.

This is **top-down dynamic programming**

# Bottom-up dynamic programming

Rather than computing table values as needed, we find an order for instances such that

- the costs of all subinstances of an instance $I$ are computed before the cost of $I$ is computed

Now just fill up the table in this order.

## Longest path

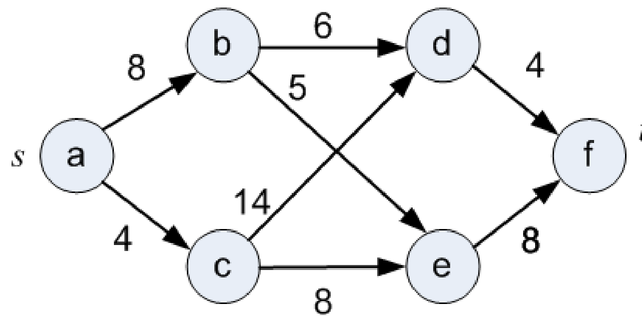
(Assume $\mathrm{topSort}(G)$ returns a sequence of nodes such that successors appear before predecessors.)

*Find the cost of the longest path from each node to $t$.*

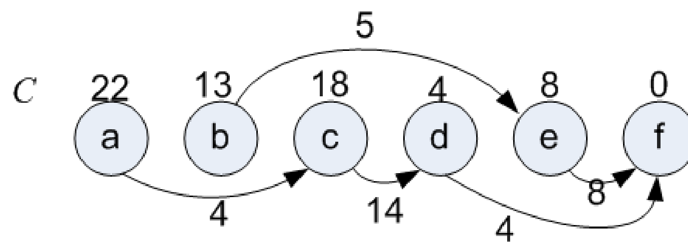
var $C : V \to \mathrm{Int} \cup \{\infty\}$
var $x : \mathrm{seq}\, V := \mathrm{topSort}(G)$
$C(t) := 0$
// Process the nodes in topological order
// invariant: for all $v$ prior to $u$ on $x$, $C(v) = \mathrm{dpl}(v, t)$
for $u \leftarrow x$, excluding $t$ do
  var $best := -\infty$
  for $v \mid u \to v$ do
    val $cost := w(u, v) + C(v)$
    if $cost > best$ then $best := cost$ end if
  end for
  $C(u) := best$
end for

[Exercise: modify this algorithm to also compute a table of optimal paths.]

For example with this graph




We fill in the table below from right to left. (Ignore the edges! They are merely illustrative.)



## Minimum edit distance

var $C : \{0, .., x.\text{length}\} \times \{0, .., y.\text{length}\} \to \text{Int}$

// First row
for $j \leftarrow \{0, .., y.\text{length}\}$ do $C(0, j) := j$ end for

// inv: The $C$ array is correctly filled for all rows that precede row $i$
for $i \leftarrow [1, .., x.\text{length}]$ do

  $C(i, 0) := i$ // First column

  // inv: The $C$ array is correctly filled for all rows that
  // precede row $i$ and all items on row $i$ that precede $j$
  for $j \leftarrow [1, .., y.\text{length}]$ do
    if $x(i - 1) = y(j - 1)$ then
      $C(i, j) := C(i - 1, j - 1)$
    else
      val $rCost := 1 + C(i - 1, j - 1)$
      val $dCost := 1 + C(i - 1, j)$
      val $iCost := 1 + C(i, j - 1)$
      $C(i, j) := \min(rCost, dCost, iCost)$
    end if
end for

Example: Changing $x$ =“FRED” to $y$ =“REND”. All possible routes.

| $x$ | $i \setminus j$ | 0 | R | 1 | E | 2 | N | 3 | D | 4 |
|---|---|---|---|---|---|---|---|---|---|---|
| = | 0 | FRED | → | RFRED | → | REFRED | → | RENFRED | → | RENDFRED |
| F | | ↓ | ↘ | ↓ | ↘ | ↓ | ↘ | ↓ | ↘ | ↓ |
| | 1 | RED | → | RRED | → | RERED | → | RENRED | → | RENDRED |
| R | | ↓ | ╲ | ↓ | ↘ | ↓ | ↘ | ↓ | ↘ | ↓ |
| | 2 | ED | → | RED | → | REED | → | RENED | → | RENDED |
| E | | ↓ | ↘ | ↓ | ╲ | ↓ | ↘ | ↓ | ↘ | ↓ |
| | 3 | D | → | RD | → | RED | → | REND | → | RENDD |
| D | | ↓ | ↘ | ↓ | ↘ | ↓ | ↘ | ↓ | ╲ | ↓ |
| | 4 | | → | R | → | RE | → | REN | → | REND |

↓ delete. → insert. ↘ replace. ╲ no change.

The cost table $C$:

| $x$ | $i \setminus j$ | 0 | R | 1 | E | 2 | N | 3 | D | 4 |
|---|---|---|---|---|---|---|---|---|---|---|
| = | 0 | 0 | → | 1 | → | 2 | → | 3 | → | 4 |
| F | | ↓ | ↘ | | ↘ | | ↘ | | ↘ | |
| | 1 | 1 | | 1 | → | 2 | → | 3 | → | 4 |
| R | | ↓ | ╲ | | ↘ | | ↘ | | ↘ | |
| | 2 | 2 | | 1 | → | 2 | → | 3 | → | 4 |
| E | | ↓ | | ↓ | ╲ | | | | | |
| | 3 | 3 | | 2 | | 1 | → | 2 | → | 3 |
| D | | ↓ | | ↓ | | ↓ | ↘ | | ╲ | |
| | 4 | 4 | | 3 | | 2 | | 2 | | 2 |

The arrows and lines are purely illustrative and show optimal routes.

## A schematic algorithm

procedure $bottomUpDynamic(R)$
let $Js$ be a sequence of instances such that
 * $R$ is in the sequence (usually it's last)
 * for any $J$ in the sequence, all its direct subinstances appear earlier in the sequence

var $C$ : a table mapping instances to costs
for $J \leftarrow Js$ do
  if $J$ is a leaf instance then
    *fill in $C(J)$ by brute force*
  else
    *compute $C(J)$ from the entries in $C$ corresponding to $J$'s subinstances*
  end if
end for
return $C(R)$

# Extracting optimal solutions (Retrospective method)

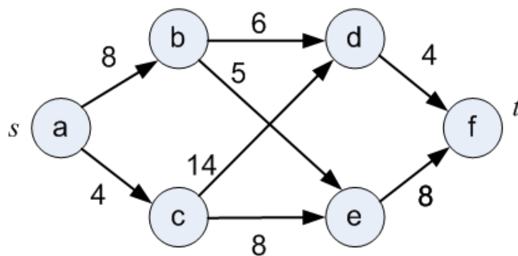
So far, we've only computed the cost of an optimal solution not the optimal solution itself.

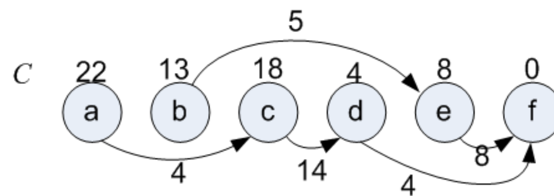
It's not hard to adapt either bottom up or top-down dynamic programming algorithms to compute optimal solutions to each subinstance. (Simply keep a second table for the solutions.)

However, usually we are only interested in finding an optimal solution to the root instance.

In that case, it may be more efficient to extract an optimal solution from the optimal cost table.

## Longest path




We reconstruct an optimal path as follows.

procedure *extractPath*( $u$ ) : $Seq[V]$

precondition, the $C$ table is correctly filled in

postcondition $result =$ a longest path from $u$ to $t$

if $u = t$ then return $[t]$

else

let $v$ be such that $u \to v$ and $C(u) = C(v) + w(u, v)$

return $[u, (u, v)]\hat{}\, extractPath(v)$

end if

end $extractPath$

[Exercise: Convert the recursion to a loop.]

## Minimum edit sequence

procedure *extractMES*( $i, j$ ) : $\mathrm{Seq}[\mathrm{Instruction}]$
  precondition: the $C$ table is correctly filled
  postcondition: $result =$ a minimum edit sequence to transform $x[0, ..i]$ to $y[0, ..j]$
  if $i = 0 = j$ then return $[\,]$
  elseif $i = 0$ then return
    extractMES( $i, j - 1$ )ˆ[insert $y(j-1)$ at $j - 1$]
  elseif $j = 0$ then return
    extractMES( $i - 1, j$ )ˆ[delete $x(i-1)$ at $0$]
  elseif $x(i-1) = y(j-1)$ then return
    extractMES( $i - 1, j - 1$ )
  else
    switch $C(i, j)$
    case $1 + C(i, j - 1)$ then
      return *extractMES*( $i, j - 1$ )
        ˆ [insert $y(j-1)$ at $j - 1$]
    case $1 + C(i - 1, j)$ then
      return *extractMES*( $i - 1, j$ )
        ˆ [delete $x(i-1)$ at $j$]
    case $1 + C(i - 1, j - 1)$ then
      return *extractMES*( $i - 1, j - 1$ )
        ˆ [replace $x(i-1)$ at $j - 1$ with $y(j-1)$]
    end switch end if end *extractMES*

[Note: If more than one case matches, any can be chosen.]

[Exercise: Remove the recursion.]

# Extracting optimal solutions (Proactive method)

Another approach is to record a little extra information while computing the cost.

For example for the MES problem. Modify the bottom up solution as follows.

Add a new table

var $H$ :: $\{0, .., x.\text{length}\} \times \{0, .., y.\text{length}\} \to \{\text{delete}, \text{insert}, \text{replace}, \text{nochange}\}$

Replace

- Replace $C(i, 0) := i$ with $C(i, 0) := i$ $H(i, 0) := \text{delete}$
- Replace $C(0, j) := j$ with $C(0, j) := j$ $H(0, j) := \text{insert}$
- Replace $C(i, j) := C(i - 1, j - 1)$ with $C(i, j) := C(i - 1, j - 1)$ $H(i, j) := \text{nochange}$
- Replace $C(i, j) := \min(rCost, dCost, iCost)$ with

  $C(i, j) := \min(rCost, dCost, iCost)$
  switch $C(i, j)$
  case $rCost$ then $H(i, j) := \text{replace}$
  case $dCost$ then $H(i, j) := \text{delete}$
  case $iCost$ then $H(i, j) := \text{insert}$
  end switch

For our example we could get the following $C/H$ tables

| $x$ | $i \diagdown j$ | 0 | | 1 | | 2 | | 3 | | 4 |
|---|---|---|---|---|---|---|---|---|---|---|
| | $y =$ | | R | | E | | N | | D | |
| $=$ | 0 | 0/? | $\rightarrow$ | 1/i | $\rightarrow$ | 2/i | $\rightarrow$ | 3/i | $\rightarrow$ | 4/i |
| F | | $\downarrow$ | $\searrow$ | | $\searrow$ | | $\searrow$ | | $\searrow$ | |
| | 1 | 1/d | | 1/r | | 2/r | | 3/r | | 4/r |
| R | | $\downarrow$ | $\diagdown$ | | $\searrow$ | | $\searrow$ | | $\searrow$ | |
| | 2 | 2/d | | 1/n | | 2/r | | 3/r | | 4/r |
| E | | $\downarrow$ | | $\downarrow$ | $\diagdown$ | | | | | |
| | 3 | 3/d | | 2/d | | 1/n | $\rightarrow$ | 2/i | $\rightarrow$ | 3/i |
| D | | $\downarrow$ | | $\downarrow$ | | $\downarrow$ | $\searrow$ | | $\diagdown$ | |
| | 4 | 4/d | | 3/d | | 2/d | | 2/r | | 2/n |

[The arrows and lines show optimal routes as encoded in the $H$ table.]

Now the $H$ array can be used to reconstruct the edit sequence working backward from the bottom right corner

var $s := [\,]$
var $(i, j) := (x.\text{length}, y.\text{length})$
while $(i, j) \neq (0, 0)$ do
  switch $H(i, j)$
  case $\text{replace}$ then
    $s :=$ [replace $x(i-1)$ at $j-1$ with $y(j-1)$]$\hat{}\,s$
    $(i, j) := (i-1, j-1)$
  case $\text{delete}$ then
    $s :=$ [delete $x(i-1)$ at $j$]$\hat{}\,s$; $i := i-1$
  case $\text{insert}$ then
    $s :=$ [insert $y(j-1)$ at $j-1$]$\hat{}\,s$; $j := j-1$
  case $\text{nochange}$ then $(i, j) := (i-1, j-1)$