Recursive search for optimal costs

Longest Path

Suppose we have a directed simple *acyclic* graph in which edges are labeled with distances.

We need to find the distance of the longest path from s to t. Each edge (u, v) has a distance $w(u, v)$.

Define $\mathrm{dlp}(u, t)$ to be the distance of the longest path from u to t: $\mathrm{dlp}(u, t) = \max_{p | u \xrightarrow{p} t} \mathrm{distance}(p)$ where

$$\mathrm{distance}([u_0, e_0, u_1, e_1, ..., e_{n-1}, u_n]) = \Sigma_{i \in \{0,..n\}} w(u_i, u_{i+1})$$

Consider $\max \emptyset$ to be $-\infty$ so $\mathrm{dlp}(u, t) = -\infty$ if there is no path from u to t.

Contract

procedure *distanceOfLongestPath*$(u : V, t : V) : \mathrm{Int} \cup \{-\infty\}$

postcondition $\mathit{result} = \mathrm{dlp}(u, t)$

Algorithmic idea: For each edge leaving u, find the length of the longest path to t that starts with that edge. Pick the best.

procedure *distanceOfLongestPath*$(u : V, t : V) : \mathrm{Int} \cup \{-\infty\}$
    postcondition $\mathit{result} = \mathrm{dlp}(u, t)$
    if $u = t$ then
        return 0;
    else
        var $\mathit{best} := -\infty$
        for $v \mid u \to v$ do
            val $\mathit{cost} := w(u, v) + \mathit{distanceOfLongestPath}(v, t)$
            if $\mathit{cost} > \mathit{best}$ then $\mathit{best} := \mathit{cost}$ end if
        end for
        return best
    end if
end *distanceOfLongestPath*

Now call *distanceOfLongestPath*(s, t).

If the answer is $-\infty$ then there is no path, else it's the distance of the longest path.

To find the longest path, we can return the longest path along side its distance

procedure $longestPath(u : V, t : V) : ((\mathrm{Int} \cup \{-\infty\}) \times \mathrm{Seq}\langle E \cup V\rangle)$

postcondition: $result = (\mathrm{c}, p)$ where $c = \mathrm{dlp}(u, t)$ and p is a path from u to t of distance c.

```
if u = t then
   return (0, [t])
else
   var bestCost : Int := −∞
   var bestPath := nil
   for v | u → v do
      val (cost,p) := longestPath(v, t)
      if cost + w(u, v) > bestCost then
            bestCost := cost + w(u, v)
            bestPath := [u, (u, v)]^p
      end if
   end for
   return (bestCost, bestPath)
end if
```

end $longestPath$

Minimum edit distance

Given two sequences, how many operations are needed to transform one into the other?

Each operation is one of

- Delete an item
- Insert an item
- Replace one item with another

Example: This edit sequence has 7 operations:

midway upon the journey of our life
in the midway of this our mortal life

insert “in” at 0

in midway upon the journey of our life
in **the** midway of this our mortal life

insert “the” at 1

in the midway **upon** the journey of our life
in the midway **of** this our mortal life

replace “upon” with “of” at 3

in the midway of **the** journey of our life
in the midway of **this** our mortal life

replace “the” with “this” at 4

in the midway of this **journey** of our life
in the midway of this our mortal life

delete “journey” at 5

in the midway of this **of** our life
in the midway of this our mortal life

delete “of” at 5

in the midway of this our life
in the midway of this our **mortal** life

insert “mortal” at 6

in the midway of this our mortal life
in the midway of this our mortal life

Is this minimal?

Applications:

- communicating and storing differences between versions of files.
- Showing the user the changes between two versions of a file.
- Finding similarity between DNA or protein sequences
- Ranking corrections for misspelled words.

Working left to right:

For any solution, there is an equivalent solution that works from left to right.

Why?

We can exchange instructions (with minor adjustments) until they are ordered from left to right.

Consider changing "FRED" to "REND". We could

"FRED" $\xrightarrow{\text{insert}(\text{'D'},4)}$ "FREDD" $\xrightarrow{\text{replace}(\text{'N'},3)}$ "FREND" $\xrightarrow{\text{delete}(0)}$ "REND"

Exchanging delete and replace

"FRED" $\xrightarrow{\text{insert}(\text{'D'},4)}$ "FREDD" $\xrightarrow{\text{delete}(0)}$ "REDD" $\xrightarrow{\text{replace}(\text{'N'},2)}$ "REND"

Exchange delete and insert

"FRED" $\xrightarrow{\text{delete}(0)}$ "RED" $\xrightarrow{\text{insert}(\text{'D'},3)}$ "REDD" $\xrightarrow{\text{replace}(\text{'N'},2)}$ "REND"

Exchange insert and replace

"FRED" $\xrightarrow{\text{delete}(0)}$ "RED" $\xrightarrow{\text{replace}(\text{'N'},2)}$ "REN" $\xrightarrow{\text{insert}(\text{'D'},3)}$ "REND"

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

$\downarrow$ deletion. $\longrightarrow$ insertion. $\searrow$ replace. $\diagdown$ no edit.

	$y =$		R		E		N		D	
x	$i\diagdown j$	0		1		2		3		4
=	0	FRED	$\longrightarrow$	RFRED	$\longrightarrow$	REFRED	$\longrightarrow$	RENFRED	$\longrightarrow$	RENDFRED
F		$\downarrow$	$\searrow$	$\downarrow$	$\searrow$	$\downarrow$	$\searrow$	$\downarrow$	$\searrow$	$\downarrow$
	1	RED	$\longrightarrow$	RRED	$\longrightarrow$	RERED	$\longrightarrow$	RENRED	$\longrightarrow$	RENDRED
R		$\downarrow$	$\diagdown$	$\downarrow$	$\searrow$	$\downarrow$	$\searrow$	$\downarrow$	$\searrow$	$\downarrow$
	2	ED	$\longrightarrow$	RED	$\longrightarrow$	REED	$\longrightarrow$	RENED	$\longrightarrow$	RENDED
E		$\downarrow$	$\searrow$	$\downarrow$	$\diagdown$	$\downarrow$	$\searrow$	$\downarrow$	$\searrow$	$\downarrow$
	3	D	$\longrightarrow$	RD	$\longrightarrow$	RED	$\longrightarrow$	REND	$\longrightarrow$	RENDD
D		$\downarrow$	$\searrow$	$\downarrow$	$\searrow$	$\downarrow$	$\searrow$	$\downarrow$	$\diagdown$	$\downarrow$
	4		$\longrightarrow$	R	$\longrightarrow$	RE	$\longrightarrow$	REN	$\longrightarrow$	REND

Note that, at entry (i, j), we have replaced $x[0, ..i]$ by $y[0, ..j]$: that is the entry is $y[0, ..j]\hat{}\,x[i, ..x.\text{length}]$.

To find the optimal cost, work backward from $y[0, ..j]$ to $x[0, ..i]$, considering the last change to be made.

Suppose x and y are sequences and $0 \le i \le x.\mathrm{length}$ and $0 \le j \le y.\mathrm{length}$

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

To change $x[0, ..i]$ to $y[0, ..j]$, there are the following possibilities.

- If $i = j = 0$, no edit is needed.
  * Cost: 0
- If $i = 0$, then j insertions is optimal.
  * Cost: j
- If $j = 0$, then i deletions is optimal.
  * Cost: i
- If $i, j > 0$, pick the cheapest of the following:
  * Edit $x[0, ..i - 1]$ to look like $y[0, ..j - 1]$
    · Cost: $\mathrm{med}(i - 1, j - 1)$
    · But only works if $x(i - 1) = y(j - 1)$,
  * Edit $x[0, ..i - 1]$ to look like $y[0, ..j - 1]$; then replace $x(j - 1)$ with $y(j - 1)$
    · Cost: $\mathrm{med}(i - 1, j - 1) + 1$
  * Edit $x[0, ..i - 1]$ to look like $y[0, ..j]$; then delete $x(j)$
    · Cost: $\mathrm{med}(i - 1, j) + 1$
  * Edit $x[0, ..i]$ to look like $y[0, ..j - 1]$; then insert $y(j - 1)$ at $j - 1$
    · Cost: $\mathrm{med}(i, j - 1) + 1$

Thus

$$\operatorname{med}(i,j) = \begin{cases} 0 & \text{if } i = 0 = j \\ j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ \min(\operatorname{med}(i-1,j-1), & \text{if } i,j > 0 \text{ and} \\ \qquad \operatorname{med}(i-1,j-1)+1, & \quad x(i-1) = y(j-1) \\ \qquad \operatorname{med}(i-1,j)+1, & \\ \qquad \operatorname{med}(i,j-1)+1)\ ) & \\ \min(\operatorname{med}(i-1,j-1)+1, & \text{if } i,j > 0 \text{ and} \\ \qquad \operatorname{med}(i-1,j)+1, & \quad x(i-1) \neq y(j-1) \\ \qquad \operatorname{med}(i,j-1)+1)\ ) & \end{cases}$$

Exercise: Show that, for all $i, j > 0$,

$$\operatorname{med}(i-1,j-1) \le \operatorname{med}(i-1,j)+1$$

and

$$\operatorname{med}(i-1,j-1) \le \operatorname{med}(i,j-1)+1$$

End Exercise.

Therefore, if $i, j > 0$ and $x(i-1) = y(j-1)$, none of the last three possibilities can cost less than simply editing $x[0,..i-1]$ to look like $y[0,..j-1]$. So:

$$\operatorname{med}(i,j) = \begin{cases} \vdots & \\ \operatorname{med}(i-1,j-1) & \text{if } i,j > 0 \text{ and} \\ & \quad x(i-1) = y(j-1) \\ \vdots & \end{cases}$$

Recall $\mathrm{med}(i, j)$ is the minimal cost to transform $x[0, ..i]$ to $y[0, ..j]$.

procedure $minEditDistance(i, j) : \mathrm{Int}$
precondition $0 \le i \le x.\mathrm{length} \wedge 0 \le j \le y.\mathrm{length}$
postcondition $result = \mathrm{med}(i, j)$
    if $i = j = 0$ **then return** 0
    elsif $i = 0$ **then return** j
    elsif $j = 0$ **then return** i
    elsif $x(i - 1) = y(j - 1)$ **then**
        return $minEditDistance(i - 1, j - 1)$
    else
        val $rCost := 1 + minEditDistance(i - 1, j - 1)$
        val $dCost := 1 + minEditDistance(i - 1, j)$
        val $iCost := 1 + minEditDistance(i, j - 1)$
        return $\min(rCost, dCost, iCost)$

Now a call to $minEditDistance(x.\mathrm{length}, y.\mathrm{length})$ computes the minimum edit distance

Exercise: Modify the algorithm so it returns a pair (c, p) where p is a list of instructions that will transform $x[0, ..i]$ to $y[0, ..j]$.

procedure $minEditSequence(i, j) : (\text{Int} \times \text{Seq}\,\langle\text{String}\rangle)$
precondition $0 \leq i \leq x.\text{length} \wedge 0 \leq j \leq y.\text{length}$
postcondition: $result = (\text{med}(i, j), p)$ where p is a list of instructions of length $\text{med}(i, j)$ that will transform $x[0, ..i]$ to $y[0, ..j]$.

For example if $x =$ “FRED” and $y =$ “REND” then $minEditSequence(i, j)$ returns

$$(2, [\text{delete}(0), \text{insert}\,(\text{‘N’}, 2)])$$

A schematic algorithm

These algorithms follow a common pattern

The optimal solution for an instance can be found by:

- Determining a set of **subinstances**
  - $*$ In the longest path problem the set of subinstances for (u, t) is
    - $\cdot$ $\{(v, t) \mid u \to v\}$ if $u \neq t$
    - $\cdot$ $\emptyset$ if $u = t$
  - $*$ In the minimum edit distance problem the set of subinstances for (i, j) is
    - $\cdot$ $\emptyset$ if $i = 0$ or $j = 0$
    - $\cdot$ $\{(i-1, j-1)\}$ if $x(i-1) = y(j-1)$
    - $\cdot$ $\{(i-1, j-1), (i, j-1), (i-1, j)\}$ otherwise
- Finding optimal solutions for the subinstances by recursion
- Finding an optimal solution from those solutions

procedure $recursiveSearch(I) : Cost \times Solution$
postcondition: $result = (c, s)$, where c is the cost of the optimal solutions and s is an optimal solution.

var $optCost$
var $optSol$
if I *is a leaf* then
    compute and return $(optCost, optSol)$
else
    let K be *the number of subinstances*
    var $optSubCost : \{0, ..K\} \to Cost$
    var $optSubSol : \{0, ..K\} \to S$
    for $k \leftarrow \{0, ..K\}$ do
        $(optSubCost(k), optSubSol(k)) :=$
        $recursiveSearch(subinstance_k)$
    end for
    compute $(optCost, optSol)$ *from* $optSubCost$ *and* $optSubSol$
end if
return $(optCost, optSol)$

Efficiency

The efficiency of recursive search is typically exponential.

If n is the depth of the recursion and b is the number of choices at each level, then the time is

$$\Theta(b^n)$$

Typically the time is $2^{\Theta(n)}$.

Computing only the cost.

In many cases, we can compute the optimal cost without computing the solution.

procedure $recursiveSearch(I) : Cost$
postcondition: $result =$the cost of the optimal solution(s).
  var $optCost$
  if I *is a leaf* then
    compute $optCost$ *directly*
  else
    let K be *the number of subinstances of* I
    var $optSubCost : \{0, ..K\} \to Cost$
    for $k \leftarrow \{0, ..K\}$ do
      $optSubCost(k) := recursiveSearch(subinstance_k)$
    end for
    compute $optCost$ *from* $optSubCost$
  end if
  return $optCost$

In many cases, the optimal solution is built from the solution to only one subinstance.

Then we don't need to store the costs of the subinstances.

procedure $recursiveSearch(I) : Cost$
postcondition: $result =$the cost of the optimal solution(s).
  var $optCost$
  if I *is a leaf* then
    compute $optCost$ *directly*
  else
    let K be *the number of subinstances of* I
    $optCost := +\infty$
    for $k \leftarrow \{0, ..K\}$ do
      var $optSubCost :=$
          $recursiveSearch(subinstance_k)$
        + *the minimum cost of transforming an optimal solution to* $subinstance_k$ *into a solution to* I
      if $optSubCost \leq optCost$ then
        $optCost := optSubCost$
      end if
    end for
  end if
  return $optCost$