Sequence Alignment

Abhiram Ranade

March 2, 2016

Similarity between strings

Similarity between strings

Input: Strings $x[1..m], y[1..n]$

Similarity between strings

Input: Strings $x[1..m], y[1..n]$
Output: Find a measure of similarity between x, y.

Similarity between strings

Input: Strings $x[1..m], y[1..n]$
Output: Find a measure of similarity between x, y.

Pairwise comparison is misleading: $x = amine, y = mines$

Similarity between strings

Input: Strings $x[1..m], y[1..n]$
Output: Find a measure of similarity between x, y.

Pairwise comparison is misleading: $x = amine, y = mines$
All $x[i], y[i]$ are different. However, the sequence similarity is obvious if you align them properly:

Similarity between strings

Input: Strings $x[1..m], y[1..n]$
Output: Find a measure of similarity between x, y.

Pairwise comparison is misleading: $x = amine, y = mines$
All $x[i], y[i]$ are different. However, the sequence similarity is obvious if you align them properly:

```
amine-
```

Similarity between strings

Input: Strings $x[1..m], y[1..n]$
Output: Find a measure of similarity between x, y.

Pairwise comparison is misleading: $x = amine, y = mines$
All $x[i], y[i]$ are different. However, the sequence similarity is obvious if you align them properly:

```
amine-
-mines
```

Similarity between strings

Input: Strings $x[1..m], y[1..n]$
Output: Find a measure of similarity between x, y.

Pairwise comparison is misleading: $x = amine, y = mines$
All $x[i], y[i]$ are different. However, the sequence similarity is obvious if you align them properly:

```
amine-
-mines
```

Now 4 pairwise comparisons yield equality!

Similarity between strings

Input: Strings $x[1..m], y[1..n]$
Output: Find a measure of similarity between x, y.

Pairwise comparison is misleading: $x = amine, y = mines$
All $x[i], y[i]$ are different. However, the sequence similarity is obvious if you align them properly:

```
amine-
-mines
```

Now 4 pairwise comparisons yield equality!

Are sequences similar after some alignment?

Sequence alignment

Sequence alignment

Alignment: Insert gaps "$-$" into x, y so that length is same.

Sequence alignment

Alignment: Insert gaps $"-"$ into x, y so that length is same.
Score $= \sum_i S(x[i], y[i])$
$S(p, q)$: 0 if $p = q$. 1 if p or q is a gap. 2 otherwise.

Sequence alignment

Alignment: Insert gaps "$-$" into x, y so that length is same.
Score $= \sum_i S(x[i], y[i])$
$S(p, q)$: 0 if $p = q$. 1 if p or q is a gap. 2 otherwise.
Goal: Find the alignment with the least score.

Sequence alignment

Alignment: Insert gaps "$-$" into x, y so that length is same.
Score $= \sum_i S(x[i], y[i])$
$S(p, q)$: 0 if $p = q$. 1 if p or q is a gap. 2 otherwise.
Goal: Find the alignment with the least score.

Similarity between x, y: least score.

Sequence alignment

Alignment: Insert gaps "$-$" into x, y so that length is same.
Score $= \sum_i S(x[i], y[i])$
$S(p, q)$: 0 if $p = q$. 1 if p or q is a gap. 2 otherwise.
Goal: Find the alignment with the least score.

Similarity between x, y: least score.

Example: Align "strip" with "tramp"

Sequence alignment

Alignment: Insert gaps "$-$" into x, y so that length is same.
Score $= \sum_i S(x[i], y[i])$
$S(p, q)$: 0 if $p = q$. 1 if p or q is a gap. 2 otherwise.
Goal: Find the alignment with the least score.

Similarity between x, y: least score.

Example: Align "strip" with "tramp"

```
stri-p
-tramp
```

Sequence alignment

Alignment: Insert gaps "$-$" into x, y so that length is same.
Score $= \sum_i S(x[i], y[i])$
$S(p, q)$: 0 if $p = q$. 1 if p or q is a gap. 2 otherwise.
Goal: Find the alignment with the least score.

Similarity between x, y: least score.

Example: Align "strip" with "tramp"

```
stri-p
-tramp
```

100210 : score at each position. Total $= 4$

Sequence alignment

Alignment: Insert gaps "$-$" into x, y so that length is same.
Score $= \sum_i S(x[i], y[i])$
$S(p, q)$: 0 if $p = q$. 1 if p or q is a gap. 2 otherwise.
Goal: Find the alignment with the least score.

Similarity between x, y: least score.

Example: Align "strip" with "tramp"

```
stri-p
-tramp
```

100210 : score at each position. Total = 4

```
strip
tramp
```

: Another alignment.
22220 : Total score = 8.

What are the decisions to be made?

What are the decisions to be made?

Where to insert the gaps.

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?
"Should a gap be inserted in position 1?"

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?
"Should a gap be inserted in position 1?"
Possible answers:

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?
"Should a gap be inserted in position 1?"
Possible answers:

A: insert gap Above

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?
"Should a gap be inserted in position 1?"
Possible answers:

A: insert gap Above

B: insert gap Below

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?
"Should a gap be inserted in position 1?"
Possible answers:

A: insert gap Above

B: insert gap Below

N: No gap.

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?
"Should a gap be inserted in position 1?"
Possible answers:

- A: insert gap Above
- B: insert gap Below
- N: No gap.

The number of decisions depend upon how many gaps we insert

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?
"Should a gap be inserted in position 1?"
Possible answers:

- A: insert gap Above
- B: insert gap Below
- N: No gap.

The number of decisions depend upon how many gaps we insert
Total number of decisions can be:

What are the decisions to be made?

Where to insert the gaps.

Is there a natural order in which we should take them?
"Should a gap be inserted in position 1?"
Possible answers:

- A: insert gap Above
- B: insert gap Below
- N: No gap.

The number of decisions depend upon how many gaps we insert
Total number of decisions can be: $m + n$

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor $=$ Best of

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.
- Best among those with last decision = A.

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.
- Best among those with last decision = A.
- Best among those with last decision = B.

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.
- Best among those with last decision = A.
- Best among those with last decision = B.

Optimal substructure? If we fix the last decision, is the problem of deciding the initial part of the descriptor equivalent to finding the best descriptor for some smaller instance?

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.
- Best among those with last decision = A.
- Best among those with last decision = B.

Optimal substructure? If we fix the last decision, is the problem of deciding the initial part of the descriptor equivalent to finding the best descriptor for some smaller instance?

Theorem: Best descriptor among those with last decision = N is Best$(x[1..i-1], y[1..j-1])||N$.

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.
- Best among those with last decision = A.
- Best among those with last decision = B.

Optimal substructure? If we fix the last decision, is the problem of deciding the initial part of the descriptor equivalent to finding the best descriptor for some smaller instance?

Theorem: Best descriptor among those with last decision = N is $\text{Best}(x[1..i-1], y[1..j-1])||N$.

Theorem: Best descriptor among those with last decision = A is $\text{Best}(x[1..i], y[1..j-1])||A$.

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.
- Best among those with last decision = A.
- Best among those with last decision = B.

Optimal substructure? If we fix the last decision, is the problem of deciding the initial part of the descriptor equivalent to finding the best descriptor for some smaller instance?

Theorem: Best descriptor among those with last decision = N is $\text{Best}(x[1..i-1], y[1..j-1])||N$.

Theorem: Best descriptor among those with last decision = A is $\text{Best}(x[1..i], y[1..j-1])||A$.

Theorem: Best descriptor among those with last decision = B is $\text{Best}(x[1..i-1], y[1..j])||B$.

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.
- Best among those with last decision = A.
- Best among those with last decision = B.

Optimal substructure? If we fix the last decision, is the problem of deciding the initial part of the descriptor equivalent to finding the best descriptor for some smaller instance?

Theorem: Best descriptor among those with last decision = N is $\text{Best}(x[1..i-1], y[1..j-1])||N$.

Theorem: Best descriptor among those with last decision = A is $\text{Best}(x[1..i], y[1..j-1])||A$.

Theorem: Best descriptor among those with last decision = B is $\text{Best}(x[1..i-1], y[1..j])||B$.

$d(i,j)$ = Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

Brute force algorithm for aligning $x[1..i], y[1..j]$:

Best descriptor = Best of

- Best among those with last decision = N.
- Best among those with last decision = A.
- Best among those with last decision = B.

Optimal substructure? If we fix the last decision, is the problem of deciding the initial part of the descriptor equivalent to finding the best descriptor for some smaller instance?

Theorem: Best descriptor among those with last decision = N is $\text{Best}(x[1..i-1], y[1..j-1])||N$.

Theorem: Best descriptor among those with last decision = A is $\text{Best}(x[1..i], y[1..j-1])||A$.

Theorem: Best descriptor among those with last decision = B is $\text{Best}(x[1..i-1], y[1..j])||B$.

$d(i,j)$ = Score for best alignment between $x[1..i], y[1..j]$

$$i = 1..m, j = 1..n$$

$$d(i,j) = \min(d(i-1,j-1) + S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$$

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$$

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$$

$d(0,j), d(i,0)$ appears in rhs.

Storing d in a table

$d(i,j)$ = Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j)$ = score to align $x[1..0], y[1..j]$, i.e. empty x

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j) =$ score to align $x[1..0], y[1..j]$, i.e. empty x $\qquad = j$

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j) =$ score to align $x[1..0], y[1..j]$, i.e. empty x $\quad = j$

$d(i,0) =$ score to align $x[1..i], y[1..0]$, i.e. empty y

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$$i = 1..m, j = 1..n$$

$$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j) =$ score to align $x[1..0], y[1..j]$, i.e. empty x $= j$

$d(i,0) =$ score to align $x[1..i], y[1..0]$, i.e. empty y $= i$

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$$i = 1..m, j = 1..n$$

$$d(i,j) = \min(d(i-1,j-1) + S(x[i], y[j]), d(i,j-1)+1, d(i-1,j)+1)$$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j) =$ score to align $x[1..0], y[1..j]$, i.e. empty x $= j$

$d(i,0) =$ score to align $x[1..i], y[1..0]$, i.e. empty y $= i$

Store $d(i,j)$ in $T[i,j]$

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j) =$ score to align $x[1..0], y[1..j]$, i.e. empty x $= j$

$d(i,0) =$ score to align $x[1..i], y[1..0]$, i.e. empty y $= i$

Store $d(i,j)$ in $T[i,j]$

$T[0,j] = j$ Base case

Storing d in a table

$d(i,j)$ = Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$d(i,j) = \min(d(i{-}1,j{-}1) + S(x[i],y[j]), d(i,j{-}1)+1, d(i{-}1,j)+1)$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j)$ = score to align $x[1..0], y[1..j]$, i.e. empty x $\quad = j$

$d(i,0)$ = score to align $x[1..i], y[1..0]$, i.e. empty y $\quad = i$

Store $d(i,j)$ in $T[i,j]$

$T[0,j] = j$ Base case

$T[i,0] = i$ Base case

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j) =$ score to align $x[1..0], y[1..j]$, i.e. empty x $= j$

$d(i,0) =$ score to align $x[1..i], y[1..0]$, i.e. empty y $= i$

Store $d(i,j)$ in $T[i,j]$

$T[0,j] = j$ Base case

$T[i,0] = i$ Base case

We can fill T left to right, top to bottom.

Storing d in a table

$d(i,j) =$ Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j) =$ score to align $x[1..0], y[1..j]$, i.e. empty x $= j$

$d(i,0) =$ score to align $x[1..i], y[1..0]$, i.e. empty y $= i$

Store $d(i,j)$ in $T[i,j]$

$T[0,j] = j$ Base case

$T[i,0] = i$ Base case

We can fill T left to right, top to bottom.

Time to fill 1 entry $= O(1)$. Total time $= O(mn)$.

Storing d in a table

$d(i,j)$ = Score for best alignment between $x[1..i], y[1..j]$

$i = 1..m, j = 1..n$

$d(i,j) = \min(d(i-1,j-1)+S(x[i],y[j]), d(i,j-1)+1, d(i-1,j)+1)$

$d(0,j), d(i,0)$ appears in rhs.

$d(0,j)$ = score to align $x[1..0], y[1..j]$, i.e. empty x $= j$

$d(i,0)$ = score to align $x[1..i], y[1..0]$, i.e. empty y $= i$

Store $d(i,j)$ in $T[i,j]$

$T[0,j] = j$ Base case

$T[i,0] = i$ Base case

We can fill T left to right, top to bottom.

Time to fill 1 entry $= O(1)$. Total time $= O(mn)$.

Space requirement: Only two rows or two columns need to be in memory.

Example: align "strip" with "tramp"

Example: align "strip" with "tramp"

		t	r	a	m	p
	0	1	2	3	4	5
s	1	2	3	4	5	6
t	2	1	2	3	4	5
r	3	2	1	2	3	4
i	4	3	2	3	4	5
p	5	4	3	4	5	4

Extracting the alignment descriptor from T

Easier to generate alignment descriptor in reverse order.

Extracting the alignment descriptor from T

Easier to generate alignment descriptor in reverse order.

1. $i, j = m, n$
2. while $i > 0$ or $j > 0$
3. if $T[i, j] = T[i-1, j-1] + S(x[i], y[j])$ then
   output "N". $i, j = i-1, j-1$
4. if $T[i, j] = T[i, j-1] + 1$ then
   output "A". $j = j-1$.
5. if $T[i, j] = T[i-1, j] + 1$ then
   output "B". $i = i-1$.
6. end while

Extracting the alignment descriptor from T

Easier to generate alignment descriptor in reverse order.

1. $i, j = m, n$
2. while $i > 0$ or $j > 0$
3. if $T[i, j] = T[i-1, j-1] + S(x[i], y[j])$ then
   output "N". $i, j = i-1, j-1$
4. if $T[i, j] = T[i, j-1] + 1$ then
   output "A". $j = j - 1$.
5. if $T[i, j] = T[i-1, j] + 1$ then
   output "B". $i = i - 1$.
6. end while

Note: if $T[*, *]$ are undefined in condition checks, then assume that those condition checks have failed.

Extracting the alignment descriptor from T

Easier to generate alignment descriptor in reverse order.

1. $i, j = m, n$
2. while $i > 0$ or $j > 0$
3. if $T[i,j] = T[i-1,j-1] + S(x[i], y[j])$ then
   output "N". $i, j = i-1, j-1$
4. if $T[i,j] = T[i,j-1] + 1$ then
   output "A". $j = j-1$.
5. if $T[i,j] = T[i-1,j] + 1$ then
   output "B". $i = i-1$.
6. end while

Note: if $T[*,*]$ are undefined in condition checks, then assume that those condition checks have failed.

Space requirement: $\theta(mn)$. Entire table is potentially needed.

Reducing space requirement to $O(m + n)$

Useful if m, n are very large.

Reducing space requirement to $O(m+n)$

Useful if m, n are very large.

Preliminary observation: Descriptor construction can be thought of as a shortest path problem

Reducing space requirement to $O(m+n)$

Useful if m, n are very large.

Preliminary observation: Descriptor construction can be thought of as a shortest path problem
Vertices: vertex (i, j) for each entry of table.

Reducing space requirement to $O(m+n)$

Useful if m, n are very large.

Preliminary observation: Descriptor construction can be thought of as a shortest path problem

Vertices: vertex (i, j) for each entry of table.

Directed Edges: (i, j) has edges from

- $(i-1, j-1)$ weight $S(x[i], y[j])$, label $=$ N.
- $(i, j-1)$ weight 1, label $=$ A.
- $(i-1, j)$ weight 1, label $=$ B.

Reducing space requirement to $O(m+n)$

Useful if m, n are very large.

Preliminary observation: Descriptor construction can be thought of as a shortest path problem

Vertices: vertex (i, j) for each entry of table.

Directed Edges: (i, j) has edges from

- $(i-1, j-1)$ weight $S(x[i], y[j])$, label $=$ N.
- $(i, j-1)$ weight 1, label $=$ A.
- $(i-1, j)$ weight 1, label $=$ B.

Lemma: $T[i, j] =$ length of shortest path from (0,0) to (i, j).

Reducing space requirement to $O(m+n)$

Useful if m, n are very large.

Preliminary observation: Descriptor construction can be thought of as a shortest path problem

Vertices: vertex (i, j) for each entry of table.

Directed Edges: (i, j) has edges from

- $(i-1, j-1)$ weight $S(x[i], y[j])$, label $=$ N.
- $(i, j-1)$ weight 1, label $=$ A.
- $(i-1, j)$ weight 1, label $=$ B.

Lemma: $T[i, j] =$ length of shortest path from (0,0) to (i, j).

Proof: Induction over $i+j$.

Reducing space requirement to $O(m+n)$

Useful if m, n are very large.

Preliminary observation: Descriptor construction can be thought of as a shortest path problem

Vertices: vertex (i, j) for each entry of table.

Directed Edges: (i, j) has edges from

- $(i-1, j-1)$ weight $S(x[i], y[j])$, label = N.
- $(i, j-1)$ weight 1, label = A.
- $(i-1, j)$ weight 1, label = B.

Lemma: $T[i, j]$ = length of shortest path from (0,0) to (i, j).

Proof: Induction over $i+j$.

Best alignment: Concatenation of labels along shortest path.

Outline

Outline

Key idea: In $O(m)$ space and $O(mn)$ time we determine a vertex in column $n/2$ through which the shortest path passes, along with the label of the edge coming into it.

Outline

Key idea: In $O(m)$ space and $O(mn)$ time we determine a vertex in column $n/2$ through which the shortest path passes, along with the label of the edge coming into it.

The determined vertex splits the shortest path into two parts.

Outline

Key idea: In $O(m)$ space and $O(mn)$ time we determine a vertex in column $n/2$ through which the shortest path passes, along with the label of the edge coming into it.

The determined vertex splits the shortest path into two parts.

Remaining vertices and labels are determined by recursing on the two parts.

Determine a "midpoint" of the shortest path

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2]$ = length of shortest path F_i from (0,0) to $(i, n/2)$.

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2]$ = length of shortest path F_i from (0,0) to $(i, n/2)$.

Consider reversed graph: Reverse all edges.

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2]$ = length of shortest path F_i from (0,0) to $(i, n/2)$.

Consider reversed graph: Reverse all edges.

- In $2m$ space we can determine length of shortest path B_i from (m, n) to all $(i, n/2)$.

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2] =$ length of shortest path F_i from (0,0) to $(i, n/2)$.

Consider reversed graph: Reverse all edges.

- In $2m$ space we can determine length of shortest path B_i from (m, n) to all $(i, n/2)$.
- Let $R_i =$ reverse of B_i.

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2] =$ length of shortest path F_i from (0,0) to $(i, n/2)$.

Consider reversed graph: Reverse all edges.

- In $2m$ space we can determine length of shortest path B_i from (m, n) to all $(i, n/2)$.
- Let $R_i =$ reverse of B_i.
- For all i, $F_i || R_i$ is a path from $(0, 0)$ to (m, n).

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2]$ = length of shortest path F_i from (0,0) to $(i, n/2)$.

Consider reversed graph: Reverse all edges.

- In $2m$ space we can determine length of shortest path B_i from (m, n) to all $(i, n/2)$.
- Let R_i = reverse of B_i.
- For all i, $F_i || R_i$ is a path from $(0, 0)$ to (m, n).
- One of these must be a shortest path from (0,0) to (m, n).

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2] =$ length of shortest path F_i from (0,0) to $(i, n/2)$.

Consider reversed graph: Reverse all edges.

- In $2m$ space we can determine length of shortest path B_i from (m, n) to all $(i, n/2)$.
- Let $R_i =$ reverse of B_i.
- For all i, $F_i || R_i$ is a path from $(0, 0)$ to (m, n).
- One of these must be a shortest path from (0,0) to (m, n).
- Find the lengths of all $F_i || R_i$, and pick the minimum.

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2]$ = length of shortest path F_i from (0,0) to $(i, n/2)$.

Consider reversed graph: Reverse all edges.

- In $2m$ space we can determine length of shortest path B_i from (m, n) to all $(i, n/2)$.
- Let R_i = reverse of B_i.
- For all i, $F_i||R_i$ is a path from $(0, 0)$ to (m, n).
- One of these must be a shortest path from (0,0) to (m, n).
- Find the lengths of all $F_i||R_i$, and pick the minimum.
- So we know the vertex $(i, n/2)$ through which the path passes.

Determine a "midpoint" of the shortest path

In $2m$ space we can determine $T[i, n/2]$ for all j.
Note: $T[i, n/2] =$ length of shortest path F_i from (0,0) to $(i, n/2)$.

Consider reversed graph: Reverse all edges.

- In $2m$ space we can determine length of shortest path B_i from (m, n) to all $(i, n/2)$.
- Let $R_i =$ reverse of B_i.
- For all i, $F_i || R_i$ is a path from $(0, 0)$ to (m, n).
- One of these must be a shortest path from (0,0) to (m, n).
- Find the lengths of all $F_i || R_i$, and pick the minimum.
- So we know the vertex $(i, n/2)$ through which the path passes.

In $O(mn)$ time and $O(m)$ space we have found the vertex through which a shortest path from (0,0) to (m, n) passes.

Determining the entire path

Determining the entire path

Recurse.

Determining the entire path

Recurse.

Time taken: $\log n$ phases, so $O(mn \log n)$

Determining the entire path

Recurse.

Time taken: $\log n$ phases, so $O(mn \log n)$
Work is reducing in each phase. With sharper analysis: $O(mn)$.

Determining the entire path

Recurse.

Time taken: $\log n$ phases, so $O(mn \log n)$
Work is reducing in each phase. With sharper analysis: $O(mn)$.

Space requirement:

Determining the entire path

Recurse.

Time taken: $\log n$ phases, so $O(mn \log n)$
Work is reducing in each phase. With sharper analysis: $O(mn)$.

Space requirement:
$O(m)$ in each phase.

Determining the entire path

Recurse.

Time taken: $\log n$ phases, so $O(mn \log n)$
Work is reducing in each phase. With sharper analysis: $O(mn)$.

Space requirement:
$O(m)$ in each phase.
But we need to remember the calculated path. So $O(n)$ more.

Determining the entire path

Recurse.

Time taken: $\log n$ phases, so $O(mn \log n)$
Work is reducing in each phase. With sharper analysis: $O(mn)$.

Space requirement:
$O(m)$ in each phase.
But we need to remember the calculated path. So $O(n)$ more.
Total $O(m + n)$.

Determining the entire path

Recurse.

Time taken: $\log n$ phases, so $O(mn \log n)$
Work is reducing in each phase. With sharper analysis: $O(mn)$.

Space requirement:
$O(m)$ in each phase.
But we need to remember the calculated path. So $O(n)$ more.
Total $O(m + n)$.

Divide and conquer + dynamic programming!

Shortest paths with negative edge weights

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work:

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Negative cycle: Directed cycle in G with total weight < 0.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Negative cycle: Directed cycle in G with total weight < 0.

Thm: Graph has a negative cycle $\Rightarrow$ shortest paths may not exist.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Negative cycle: Directed cycle in G with total weight < 0.

Thm: Graph has a negative cycle $\Rightarrow$ shortest paths may not exist.
Proof: Start at s, go to the negative cycle, traverse it several times, then to to t. Path weight can be made as small as we want.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Negative cycle: Directed cycle in G with total weight < 0.

Thm: Graph has a negative cycle $\Rightarrow$ shortest paths may not exist.
Proof: Start at s, go to the negative cycle, traverse it several times, then to to t. Path weight can be made as small as we want.

Thm: No negative cycles $\Rightarrow$ s, t shortest paths must exist.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Negative cycle: Directed cycle in G with total weight < 0.

Thm: Graph has a negative cycle $\Rightarrow$ shortest paths may not exist.
Proof: Start at s, go to the negative cycle, traverse it several times, then to to t. Path weight can be made as small as we want.

Thm: No negative cycles $\Rightarrow$ s, t shortest paths must exist.
Proof: Non simple path will not be shortest because we can remove the cycle and reduce the weight.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Negative cycle: Directed cycle in G with total weight < 0.

Thm: Graph has a negative cycle $\Rightarrow$ shortest paths may not exist.
Proof: Start at s, go to the negative cycle, traverse it several times, then to to t. Path weight can be made as small as we want.

Thm: No negative cycles $\Rightarrow$ s, t shortest paths must exist.
Proof: Non simple path will not be shortest because we can remove the cycle and reduce the weight.
Shortest path must have length at most $n - 1$.

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Negative cycle: Directed cycle in G with total weight < 0.

Thm: Graph has a negative cycle $\Rightarrow$ shortest paths may not exist.
Proof: Start at s, go to the negative cycle, traverse it several times, then to to t. Path weight can be made as small as we want.

Thm: No negative cycles $\Rightarrow$ s, t shortest paths must exist.
Proof: Non simple path will not be shortest because we can remove the cycle and reduce the weight.
Shortest path must have length at most $n - 1$.

n = number of vertices, m = number of edges

Shortest paths with negative edge weights

Input: Graph G with positive or negative edge weights. Nodes s, t.
Output: Least weight path from s to t.

Dijkstra does not work: $w_{st} = 5$, $w_{su} = 6$, $w_{ut} = -3$.

Negative cycle: Directed cycle in G with total weight < 0.

Thm: Graph has a negative cycle $\Rightarrow$ shortest paths may not exist.
Proof: Start at s, go to the negative cycle, traverse it several times, then to to t. Path weight can be made as small as we want.

Thm: No negative cycles $\Rightarrow$ s, t shortest paths must exist.
Proof: Non simple path will not be shortest because we can remove the cycle and reduce the weight.
Shortest path must have length at most $n - 1$.

n = number of vertices, m = number of edges

Only finite number of paths to consider.

An attempt at a brute force recursive algorithm

An attempt at a brute force recursive algorithm

Decisions that we make in designing an algorithm: What is the last node in the path?

An attempt at a brute force recursive algorithm

Decisions that we make in designing an algorithm: What is the last node in the path?

some out-neighbour of s.

An attempt at a brute force recursive algorithm

Decisions that we make in designing an algorithm: What is the last node in the path?
some out-neighbour of s.

Optimal substructure: the entire path from s to t must consist of the edge from s to some neighbour si, and a shortest path from si to t.

An attempt at a brute force recursive algorithm

Decisions that we make in designing an algorithm: What is the last node in the path?
some out-neighbour of s.

Optimal substructure: the entire path from s to t must consist of the edge from s to some neighbour si, and a shortest path from si to t.

```
sPath(s,t){
 for each out-neighbour si of s
   Pi = s || sPath(si,t)
 end for

 return the shortest among Pi
}
```

An attempt at a brute force recursive algorithm

Decisions that we make in designing an algorithm: What is the last node in the path?
some out-neighbour of s.

Optimal substructure: the entire path from s to t must consist of the edge from s to some neighbour si, and a shortest path from si to t.

```
sPath(s,t){
 for each out-neighbour si of s
   Pi = s || sPath(si,t)
 end for

 return the shortest among Pi
}
```

Not proper recursion: Instances on which recursive calls are made are not "simpler" than original instance.

Make the recursion work!

Make the recursion work!

```
sPath(i,v){  // shortest v-t path of at most i edges
  if i = 0 and v = t then return null
  if i = 0 and v != t then return "invalid"

  P0 = sPath(i-1,v)  // if length(shortest path) < i
  for each out-neighbour vj of v
    Pi = v || sPath(i-1,vj)
  end for

  return shortest valid path among Pi.
}
```

Make the recursion work!

```
sPath(i,v){  // shortest v-t path of at most i edges
  if i = 0 and v = t then return null
  if i = 0 and v != t then return "invalid"

  P0 = sPath(i-1,v)  // if length(shortest path) < i
  for each out-neighbour vj of v
    Pi = v || sPath(i-1,vj)
  end for

  return shortest valid path among Pi.
}
```

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

Make the recursion work!

```
sPath(i,v){  // shortest v-t path of at most i edges
  if i = 0 and v = t then return null
  if i = 0 and v != t then return "invalid"

  P0 = sPath(i-1,v)  // if length(shortest path) < i
  for each out-neighbour vj of v
    Pi = v || sPath(i-1,vj)
  end for

  return shortest valid path among Pi.
}
```

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$= 0$ if $i = 0$ and $v = t$

Make the recursion work!

```
sPath(i,v){  // shortest v-t path of at most i edges
  if i = 0 and v = t then return null
  if i = 0 and v != t then return "invalid"

  P0 = sPath(i-1,v)  // if length(shortest path) < i
  for each out-neighbour vj of v
    Pi = v || sPath(i-1,vj)
  end for

  return shortest valid path among Pi.
}
```

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$= 0$ if $i = 0$ and $v = t$

$= \infty$ if $i = 0$ and $v \neq t$

Make the recursion work!

```
sPath(i,v){  // shortest v-t path of at most i edges
  if i = 0 and v = t then return null
  if i = 0 and v != t then return "invalid"

  P0 = sPath(i-1,v)  // if length(shortest path) < i
  for each out-neighbour vj of v
    Pi = v || sPath(i-1,vj)
  end for

  return shortest valid path among Pi.
}
```

$d(i, v)$ = length of shortest $v - t$ path of at most i edges.

$= 0$ if $i = 0$ and $v = t$

$= \infty$ if $i = 0$ and $v \neq t$

$= \min(d(i-1, v), \min_{u|v\sim u}(w_{vu} + d(i-1, u)))$ otherwise

The table

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

The table

$d(i, v)$ = length of shortest $v - t$ path of at most i edges.
$= 0$ if $i = 0$ and $v = t$

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$= 0$ if $i = 0$ and $v = t$

$= \infty$ if $i = 0$ and $v \neq t$

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$= 0$ if $i = 0$ and $v = t$

$= \infty$ if $i = 0$ and $v \neq t$

$= \min(d(i-1, v), \min_{u | v \sim u}(w_{vu} + d(i-1, u)))$ otherwise

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$$
\begin{aligned}
&= 0 && \text{if } i = 0 \text{ and } v = t\\
&= \infty && \text{if } i = 0 \text{ and } v \neq t\\
&= \min(d(i-1, v), \min_{u|v\sim u}(w_{vu} + d(i-1, u))) && \text{otherwise}
\end{aligned}
$$

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

The table

$d(i, v)$ = length of shortest $v - t$ path of at most i edges.

$= 0$ if $i = 0$ and $v = t$

$= \infty$ if $i = 0$ and $v \neq t$

$= \min(d(i-1, v), \min_{u|v \sim u}(w_{vu} + d(i-1, u)))$ otherwise

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$$
\begin{aligned}
&= 0 && \text{if } i = 0 \text{ and } v = t \\
&= \infty && \text{if } i = 0 \text{ and } v \neq t \\
&= \min(d(i-1, v), \min_{u|v \sim u}(w_{vu} + d(i-1, u))) && \text{otherwise}
\end{aligned}
$$

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$
$T[0, v] = \infty$ if $v \neq t$

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.
$= 0$ if $i = 0$ and $v = t$
$= \infty$ if $i = 0$ and $v \neq t$
$= \min(d(i-1, v), \min_{u|v \sim u}(w_{vu} + d(i-1, u)))$ otherwise

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$
$T[0, v] = \infty$ if $v \neq t$
for $i = 1..n-1$

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.
$= 0$ if $i = 0$ and $v = t$
$= \infty$ if $i = 0$ and $v \neq t$
$= \min(d(i-1, v), \min_{u|v\sim u}(w_{vu} + d(i-1, u)))$ otherwise

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$
$T[0, v] = \infty$ if $v \neq t$
for $i = 1..n - 1$
    for $v = 1..n$ except t

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$= 0$ if $i = 0$ and $v = t$

$= \infty$ if $i = 0$ and $v \neq t$

$= \min(d(i-1, v), \min_{u|v\sim u}(w_{vu} + d(i-1, u)))$ otherwise

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$
$T[0, v] = \infty$ if $v \neq t$
for $i = 1..n-1$
    for $v = 1..n$ except t
        $T[i, v] = \min(T[i-1, v], \min_{u|v\sim u}(w_{vu} + T[i-1, u]))$

The table

$d(i, v)$ = length of shortest $v - t$ path of at most i edges.

$$
\begin{aligned}
&= 0 && \text{if } i = 0 \text{ and } v = t \\
&= \infty && \text{if } i = 0 \text{ and } v \neq t \\
&= \min(d(i-1, v), \min_{u|v \sim u}(w_{vu} + d(i-1, u))) && \text{otherwise}
\end{aligned}
$$

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$
$T[0, v] = \infty$ if $v \neq t$
for $i = 1..n-1$
    for $v = 1..n$ except t
        $T[i, v] = \min(T[i-1, v], \min_{u|v \sim u}(w_{vu} + T[i-1, u]))$
    end for

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$$
\begin{aligned}
&= 0 && \text{if } i = 0 \text{ and } v = t \\
&= \infty && \text{if } i = 0 \text{ and } v \neq t \\
&= \min(d(i-1, v), \min_{u|v \sim u}(w_{vu} + d(i-1, u))) && \text{otherwise}
\end{aligned}
$$

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$
$T[0, v] = \infty$ if $v \neq t$
for $i = 1..n-1$
    for $v = 1..n$ except t
        $T[i, v] = \min(T[i-1, v], \min_{u|v \sim u}(w_{vu} + T[i-1, u]))$
    end for
end for

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$= 0$ if $i = 0$ and $v = t$

$= \infty$ if $i = 0$ and $v \neq t$

$= \min(d(i-1, v), \min_{u|v\sim u}(w_{vu} + d(i-1, u)))$ otherwise

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$
$T[0, v] = \infty$ if $v \neq t$
for $i = 1..n-1$
    for $v = 1..n$ except t
        $T[i, v] = \min(T[i-1, v], \min_{u|v\sim u}(w_{vu} + T[i-1, u]))$
    end for
end for

Innermost loop time $= \sum_v outdegree(v) = O(m)$.

The table

$d(i, v) =$ length of shortest $v - t$ path of at most i edges.

$$
\begin{aligned}
&= 0 && \text{if } i = 0 \text{ and } v = t \\
&= \infty && \text{if } i = 0 \text{ and } v \neq t \\
&= \min(d(i-1, v), \min_{u|v\sim u}(w_{vu} + d(i-1, u))) && \text{otherwise}
\end{aligned}
$$

i need be at most $n - 1$. So we need a table $T[1..n-1, 1..n]$

$T[0, t] = 0$
$T[0, v] = \infty$ if $v \neq t$
for $i = 1..n-1$
    for $v = 1..n$ except t
        $T[i, v] = \min(T[i-1, v], \min_{u|v\sim u}(w_{vu} + T[i-1, u]))$
    end for
end for

Innermost loop time $= \sum_v \mathit{outdegree}(v) = O(m)$.
Total time $= O(mn)$.