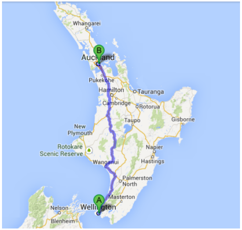
Algorithm Design and Analysis

Day 8
Dynamic Programming

AUT, 2015

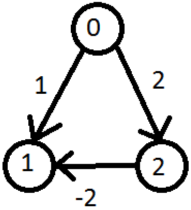
Part I: Shortest Path, Revisited



Shortest Path with Potentially Negative Edges

Recap: Being Greedy is Risky

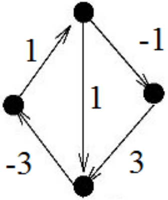
Dijkstra's algorithm does not work for weighted graph where the weights could be negative.



Greedy choice does not work here.

Shortest Path with Potentially Negative Edges

Observation



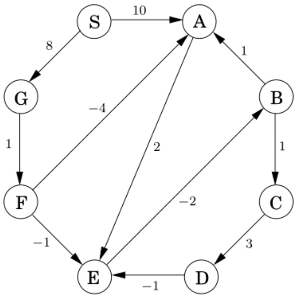
- If there is a negative cycle (cycle of negative weight), then the problem does not make sense.
- For simplicity let's first assume there is no negative cycle.

Paths with Bounded Length

Notation

Let $d_k(u)$ denote the length of the shortest path from S to u that uses at most k edges.

Example:



$d_0(A) = \infty, d_1(A) = 10, d_2(A) = 10, d_3(A) = 3$
$d_0(E) = d_1(E) = \infty, d_2(E) = 12, d_3(E) = 8, d_4(E) = 7$

Paths with Bounded Length

Fact

Suppose G does not contain a negative cycle. Then the distance from S to u is $d_{n-1}(u)$ for every $u \in V$.

Why?

Paths with Bounded Length

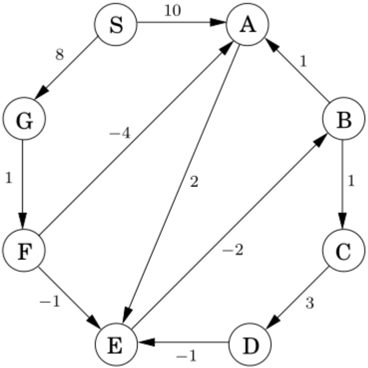
Fact

Suppose G does not contain a negative cycle. Then the distance from S to u is $d_{n-1}(u)$ for every $u \in V$.

Why? Take a path from s to u of length $> n - 1$.
It must contain $> n$ nodes.
$\Rightarrow$ Some node has been repeated.
$\Rightarrow$ There is a cycle in the path.
$\Rightarrow$ But then we can reduce the length by removing this cycle.

Computing Distances

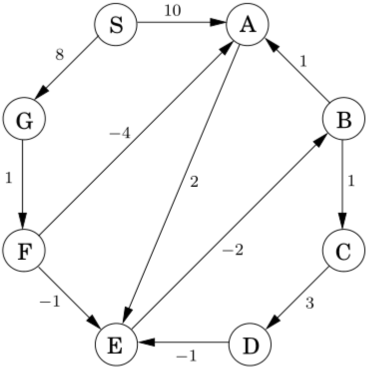
Computing distances is reduced to computing $d_{n-1}(u)$.



	k							
Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Computing Distances

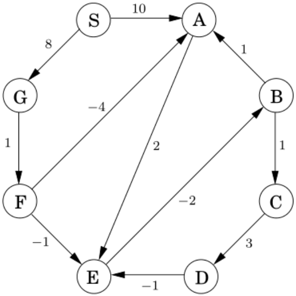
Computing distances is reduced to computing $d_{n-1}(u)$.



distances

	k							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Computing Distances

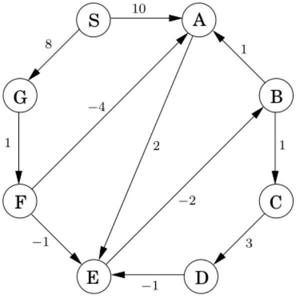


Node	0
S	0
A	∞
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

Computing $d_0(u)$

- $d_0(s) = 0$
- $d_0(u) = \infty$ for every $u \neq s$

Computing Distances

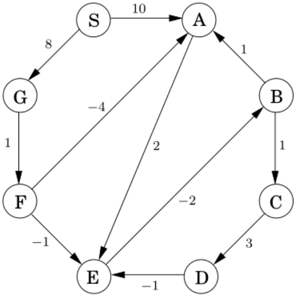


Node	0	1
S	0	0
A	∞	10
B	∞	∞
C	∞	∞
D	∞	∞
E	∞	∞
F	∞	∞
G	∞	8

Computing $d_1(u)$

- $d_1(s) = 0$
- $d_1(u) = w(s, u)$ for every out-neighbour u of s
- $d_1(u) = \infty$ for all other nodes

Computing Distances

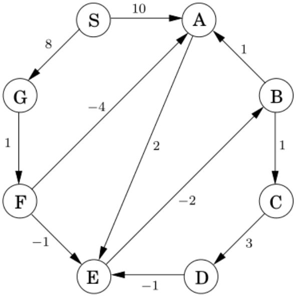


Node	0	1	2
S	0	0	0
A	∞	10	10
B	∞	∞	∞
C	∞	∞	∞
D	∞	∞	∞
E	∞	∞	12
F	∞	∞	9
G	∞	8	8

Computing $d_2(u)$

How can we tell that $d_2(E) = 12$?
$d_1(A) = 10$ and $w(A, E) = 2$
So $d_2(E) = d_1(A) + w(A, E)$

Computing Distances

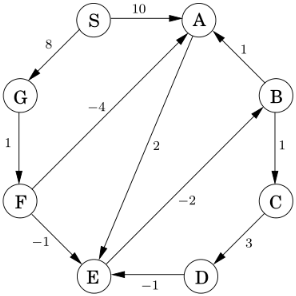


Node	0	1	2	3
S	0	0	0	0
A	∞	10	10	5
B	∞	∞	∞	10
C	∞	∞	∞	∞
D	∞	∞	∞	∞
E	∞	∞	12	8
F	∞	∞	9	9
G	∞	8	8	8

Computing $d_3(u)$

How can we tell that $d_3(A) = 5$?
$d_2(F) = 9$ and $w(F, A) = -4$
So $d_3(A) = d_2(F) + w(F, A)$

Computing Distances

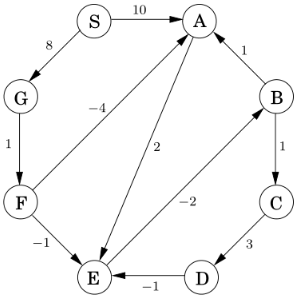


	k							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Computing $d_3(u)$

How can we tell that $d_3(B) = 10$?
$d_2(E) = 12$ and $w(E, B) = -2$
So $d_3(B) = d_2(E) + w(E, B)$

Computing Distances



	k							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

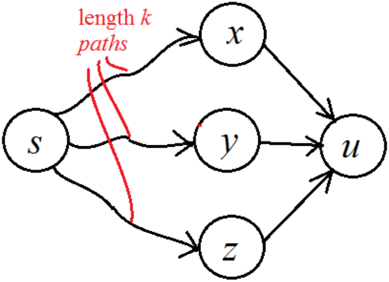
Computing $d_3(u)$

How can we tell that $d_3(E) = 8$?

$d_2(F) = 9$ and $w(F, E) = -1$; $d_2(A) = 10$ and $w(A, F) = 2$

So $d_3(B) = d_2(F) + w(F, E)$

Computing Distances



Computing $d_{k+1}(u)$

$d_{k+1}(u)$ is the smallest among

$$d_k(u), d_k(x) + w(x,u), d_k(y) + w(y,u), d_k(z) + w(z,u)$$

$\Rightarrow d_{k+1}(u) = \min\{d_k(u), \min\{d_k(v) + w(v,u) \mid (v,u) \in E\}\}$

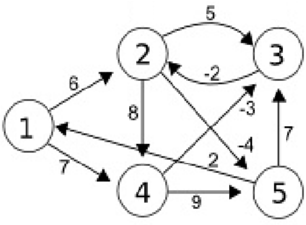
Bellman-Ford Algorithm

Bellman-Ford(G, s)

INPUT: A graph G (without negative cycle) and starting node s
OUTPUT: $d(u)$ for every node u denoting distance from s to u
$d(s) \leftarrow 0$, $d(v) \leftarrow \infty$ for all other v

```
for i = 1 to n − 1 do :
    for u ∈ V do
        d′(u) ← d(u)
        for (v, u) ∈ E do
            d′(u) ← min{d′(u), d(v) + w(v, u)}
    Replace d by d′
```

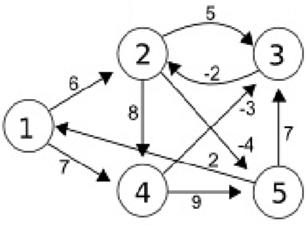
Bellman-Ford Algorithm



Iteration 1:

u	1	2	3	4	5
$d(u)$	0	∞	∞	∞	∞
$d'(u)$	0	6	∞	7	∞

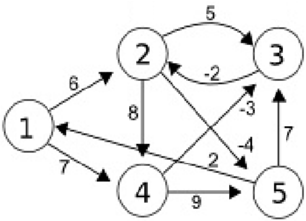
Bellman-Ford Algorithm



Iteration 1:

u	1	2	3	4	5
$d(u)$	0	6	∞	7	∞
$d'(u)$	0	6	∞	7	∞

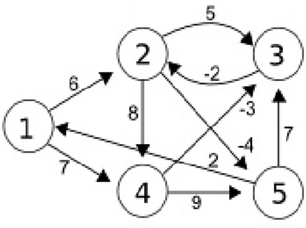
Bellman-Ford Algorithm



Iteration 2:

u	1	2	3	4	5
$d(u)$	0	6	∞	7	∞
$d'(u)$	0	6	4	7	2

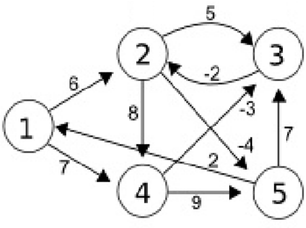
Bellman-Ford Algorithm



Iteration 2:

u	1	2	3	4	5
$d(u)$	0	6	4	7	2
$d'(u)$	0	6	4	7	2

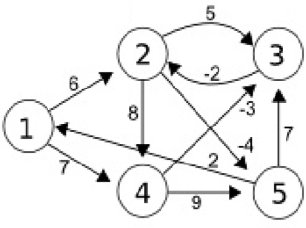
Bellman-Ford Algorithm



Iteration 3:

u	1	2	3	4	5
$d(u)$	0	6	4	7	2
$d'(u)$	0	2	4	7	-2

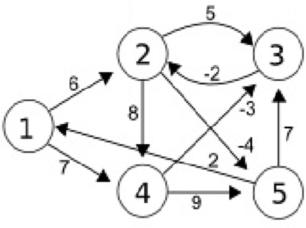
Bellman-Ford Algorithm



Iteration 3:

u	1	2	3	4	5
$d(u)$	0	2	4	7	-2
$d'(u)$	0	2	4	7	-2

Bellman-Ford Algorithm



Iteration 4:

u	1	2	3	4	5
$d(u)$	0	2	4	7	-2
$d'(u)$	0	2	4	7	-2

Bellman-Ford Algorithm: Correctness

Fact 1.

Suppose G has no negative cycle. After running Bellman-Ford(G, s), $d(u)$ is the distance from s to u for every $u \in V$.

Bellman-Ford Algorithm: Correctness

Fact 1.

Suppose G has no negative cycle. After running Bellman-Ford(G, s), $d(u)$ is the distance from s to u for every $u \in V$.

Proof. We prove the following loop invariant by induction on i:

After i rounds of the outmost for-loop, $d(u) = d_i(u)$.

Base case: $i = 0$. Before the loop, the statement is clear.

Inductive step: Suppose after i rounds, $d(u) = d_i(u)$ for every $u \in V$.
Note $d_{i+1}(u) = \min\{d_i(u), \min\{d_i(v) + w(v, u) \mid (v, u) \in E\}\}$.
Therefore after the $(i + 1)$th round, $d(u) = d_{i+1}(u)$. $\square$

Bellman-Ford Algorithm: Complexity

Fact 2.

Suppose G has no negative cycle. Bellman-Ford(G, s) computes the shortest distance from s to all other nodes in the graph G in time $\Theta(mn)$.

Proof.

- There are $n - 1$ iterations
- At each iteration, we process every incoming edge for every node
  $\Rightarrow$ We examine every edge in the graph (exactly once).

Therefore the total running time is $(n - 1) \times m$. $\square$

All-Pair Shortest Path

Dijkstra's and Bell-Ford algorithm both solves Single-Source Shortest Path Problem.

All-Pair Shortest Path Problem

Compute the shortest distance between any pair of nodes in G.

All-Pair Shortest Path

Dijkstra's and Bell-Ford algorithm both solves Single-Source Shortest Path Problem.

All-Pair Shortest Path Problem

Compute the shortest distance between any pair of nodes in G.

Solution: Run the single-source algorithm n times, each time from a different node!
$\Rightarrow O(n^2 m)$ time

All-Pair Shortest Path

Dijkstra's and Bell-Ford algorithm both solves Single-Source Shortest Path Problem.

All-Pair Shortest Path Problem

Compute the shortest distance between any pair of nodes in G.

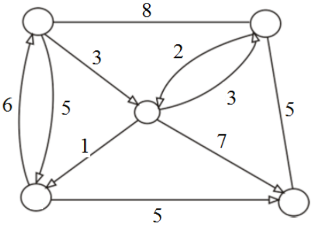
Solution: Run the single-source algorithm n times, each time from a different node!
$\Rightarrow O(n^2m)$ time

Note:

- m could be as large as $\Theta(n^2)$.
- So Bellman-Ford algorithm would runs in $O(n^4)$.
- We would like a faster algorithm for this problem.

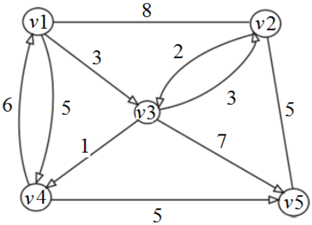
Floyd-Warshall Algorithm

Floyd-Warshall algorithm



Floyd-Warshall algorithm

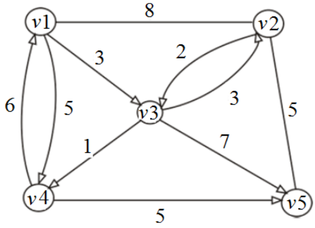
- Label all nodes using $v_1, v_2, \ldots, v_n$.



Floyd-Warshall Algorithm

Floyd-Warshall algorithm

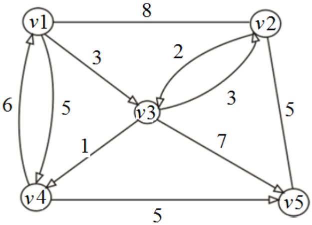
- Label all nodes using $v_1, v_2, \ldots, v_n$.
- Define $f_k(i, j)$ as the length of the shortest path between v_i, v_j that uses only $v_1, \ldots, v_k$ as intermediate nodes.



Floyd-Warshall Algorithm

Floyd-Warshall algorithm

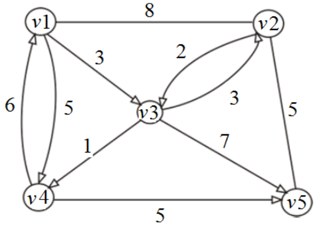
- Label all nodes using $v_1, v_2, \ldots, v_n$.
- Define $f_k(i, j)$ as the length of the shortest path between v_i, v_j that uses only $v_1, \ldots, v_k$ as intermediate nodes.
- **Fact**: $f_n(i, j)$ is the distance from v_i to v_j.



Floyd-Warshall Algorithm

Floyd-Warshall algorithm

- Label all nodes using $v_1, v_2, \ldots, v_n$.
- Define $f_k(i, j)$ as the length of the shortest path between v_i, v_j that uses only $v_1, \ldots, v_k$ as intermediate nodes.
- **Fact**: $f_n(i, j)$ is the distance from v_i to v_j.



Example: Going from 1 to 5:
Able to pass through v_1, v_2: $f_2(1,5) = 13$
Able to pass through v_1, v_2, v_3: $f_3(1,5) = 10$
Able to pass through v_1, v_2, v_3, v_4: $f_4(1,5) = 9$

Floyd-Warshall Algorithm

Suppose G does not contain a negative cycle.

Computing $f_0(i, j)$

$f_0(i, j) = w(v_i, v_j)$

Floyd-Warshall Algorithm

Suppose G does not contain a negative cycle.

Computing $f_0(i,j)$

$f_0(i,j) = w(v_i, v_j)$

Computing $f_{k+1}(i,j)$

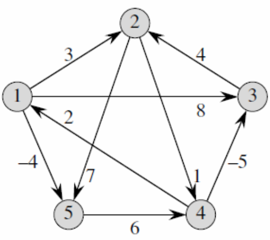
- Optimal Substructure: Suppose $u \rightsquigarrow v_{k+1} \rightsquigarrow v$ is a shortest path from u, v that only uses $v_1, \ldots, v_{k+1}$ as intermediate nodes, then $u \rightsquigarrow v_{k+1}$ and $v_{k+1} \rightsquigarrow v$ are shortest paths from u to v_{k+1} and from v_{k+1} to v using only $v_1, \ldots, v_k$ as intermediate nodes.
- Therefore we have
$$f_{k+1}(i,j) = \min\{f_k(i,j), f_k(i,k+1) + f_k(k+1,v)\}$$

Floyd-Warshall Algorithm

Floyd-Warshall(G)

INPUT: A graph G (without negative cycle)
OUTPUT: $f(i,j)$ for any i,j denoting distance from v_i to v_j
Create 2-dim arrays f, f'
$f(i,j) \leftarrow w(v_i, v_j)$ for all i,j

```
for k = 1..n do
    for i = 1 to n do
        for j = 1 to n do
            f'(i,j) ← min{f(i,j), f(i,k) + f(k,j)}
    Replace f by f'
```



Initial Stage

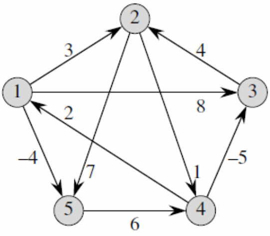
$$f = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithm

Floyd-Warshall(G)

INPUT: A graph G (without negative cycle)
OUTPUT: $f(i,j)$ for any i,j denoting distance from v_i to v_j
Create 2-dim arrays f, f'
$f(i,j) \leftarrow w(v_i, v_j)$ for all i,j

```
for k = 1..n do
    for i = 1 to n do
        for j = 1 to n do
            f'(i,j) ← min{f(i,j), f(i,k) + f(k,j)}
    Replace f by f'
```



Stage 1

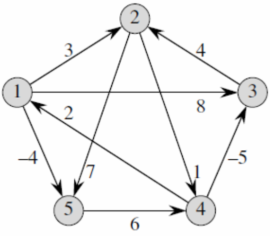
$$f = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithm

Floyd-Warshall(G)

INPUT: A graph G (without negative cycle)
OUTPUT: $f(i,j)$ for any i,j denoting distance from v_i to v_j
Create 2-dim arrays f, f'
$f(i,j) \leftarrow w(v_i, v_j)$ for all i, j

```
for k = 1..n do
    for i = 1 to n do
        for j = 1 to n do
            f'(i,j) ← min{f(i,j), f(i,k) + f(k,j)}
    Replace f by f'
```



Stage 2

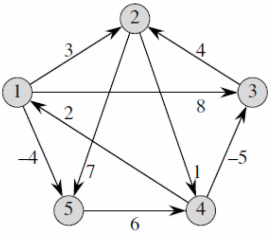
$$f = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithm

Floyd-Warshall(G)

INPUT: A graph G (without negative cycle)
OUTPUT: $f(i,j)$ for any i,j denoting distance from v_i to v_j
Create 2-dim arrays f, f'
$f(i,j) \leftarrow w(v_i, v_j)$ for all i,j

```
for k = 1..n do
    for i = 1 to n do
        for j = 1 to n do
            f'(i,j) ← min{f(i,j), f(i,k) + f(k,j)}
    Replace f by f'
```



Stage 3

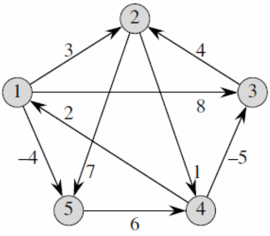
$$f = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithm

Floyd-Warshall(G)

INPUT: A graph G (without negative cycle)
OUTPUT: $f(i,j)$ for any i,j denoting distance from v_i to v_j
Create 2-dim arrays f, f'
$f(i,j) \leftarrow w(v_i, v_j)$ for all i,j

```
for k = 1..n do
    for i = 1 to n do
        for j = 1 to n do
            f'(i,j) ← min{f(i,j), f(i,k) + f(k,j)}
    Replace f by f'
```



Stage 4

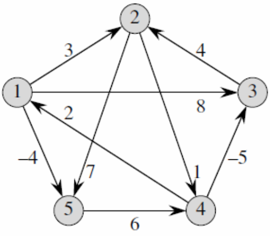
$$f = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithm

Floyd-Warshall(G)

INPUT: A graph G (without negative cycle)
OUTPUT: $f(i, j)$ for any i, j denoting distance from v_i to v_j
Create 2-dim arrays f, f'
$f(i, j) \leftarrow w(v_i, v_j)$ for all i, j

```
for k = 1..n do
    for i = 1 to n do
        for j = 1 to n do
            f'(i, j) ← min{f(i, j), f(i,k) + f(k, j)}
    Replace f by f'
```



Stage 5

$$f = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Floyd-Warshall Algorithm

Time Complexity : $O(n^3)$

Floyd-Warshall Algorithm

Time Complexity : $O(n^3)$

Further Questions

- How can we modify Floyd-Warshall Algorithm to give us the shortest paths, rather than the distances?
- What if the graph has a negative cycle? Can Floyd-Warshall Algorithm detect it?

Richard Bellman
``Dynamic Programming'' 1957

Robert Floyd
``Assigning Meaning to Programs''
1967

Shortest Paths, A Summary

- Single-Source Positive Weights: Dijkstra's algorithm

  Complexity: Depends on the priority queue implementation

  - List $O(n^2)$
  - Binary /Binomial Heap $O((n+m)\log n)$
  - Fibonacci Heap $O(m + n\log n)$

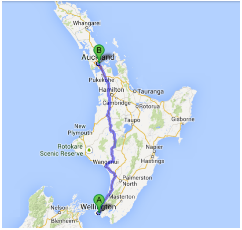
- Single-Source Positive/Negative Weights: Bellman-Ford algorithm

  Complexity: $O(nm)$

- All-Pair: Floyd-Warshall algorithm

  Complexity: $O(n^3)$

Part II: Dynamic Programming as an Algorithm



Optimization

Recall

An optimization problem contains a solution set where each solution has a value. The problem asks to find the solution with the maximal/minimal value (The optimal solution).

Examples of Optimization Problem

- Shortest Path
- Minimum Spanning Tree
- Knapsack Problem
- Sorting

Optimization

Recall

An optimization problem contains a solution set where each solution has a value. The problem asks to find the solution with the maximal/minimal value (The optimal solution).

Examples of Optimization Problem

- Shortest Path
- Minimum Spanning Tree
- Knapsack Problem
- Sorting (Reformulated): Arrange a collection of n numbers into a sequence
$$a_1, a_2, \ldots, a_n$$
where the length of the longest increasing subsequence is maximized.

Sorting VS Shortest Path

Sorting VS Shortest Path

- Similarity: [Optimal Substructure]
  - Sorting: In a sorted array, any subarray is also sorted
  - SP: In a shortest path, any segment is also a shortest path

Sorting VS Shortest Path

- Similarity: [Optimal Substructure]
  - Sorting: In a sorted array, any subarray is also sorted
  - SP: In a shortest path, any segment is also a shortest path

  $\Rightarrow$ both problems can be solved by "division".

Sorting VS Shortest Path

- Similarity: [Optimal Substructure]

  - Sorting: In a sorted array, any subarray is also sorted

  - SP: In a shortest path, any segment is also a shortest path

  $\Rightarrow$ both problems can be solved by "division".

- Difference: Suppose we divide the problem into subproblems

  - Sorting: The subproblems are completely independent.

    Hence a top-down algorithm is suitable

    $\Rightarrow$ Divide and Conquer

  - SP: The subproblems overlap.

    Hence a bottom-up algorithm is suitable

    $\Rightarrow$ Dynamic Programming

Dynamic Programming

Dynamic Programming

- Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems.
- It is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure.
- Dynamic programming solves the subproblems from small to large to avoid duplication

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $d_0, d_1, d_2, \ldots, d_{n-1}$

   Parameter: Number of edges used in the shortest path.

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $d_0, d_1, d_2, \ldots, d_{n-1}$

   Parameter: Number of edges used in the shortest path.

2. Handle the base case

   $d_0(u) = 0$ if $u = s$; $d_0(u) = \infty$ if $u \neq s$.

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $d_0, d_1, d_2, \ldots, d_{n-1}$

   Parameter: Number of edges used in the shortest path.

2. Handle the base case

   $d_0(u) = 0$ if $u = s$; $d_0(u) = \infty$ if $u \neq s$.

3. Write a recurrence for larger subproblems

   $$d_{k+1}(u) = \min\{d_k(u), \min\{d_k(v) + w(v,u) \mid (v,u) \in E\}\}$$

Example: Single-Source Shortest Path

Bellman-Ford algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $d_0, d_1, d_2, \ldots, d_{n-1}$

   Parameter: Number of edges used in the shortest path.

2. Handle the base case

   $d_0(u) = 0$ if $u = s$; $d_0(u) = \infty$ if $u \neq s$.

3. Write a recurrence for larger subproblems

   $d_{k+1}(u) = \min\{d_k(u), \min\{d_k(v) + w(v,u) \mid (v,u) \in E\}\}$

4. Fill the table of partial solutions in a bottom-up way

   Start from d_0, then compute $d_1, d_2, \ldots, d_{n-1}$

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $f_k(i, j)$ for all $1 \leq k \leq n$.

   Parameter: Indices of nodes used as intermediate nodes.

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $f_k(i, j)$ for all $1 \leq k \leq n$.

   Parameter: Indices of nodes used as intermediate nodes.

2. Handle the base case

   $f_0(i, j) = w(v_i, v_j)$.

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $f_k(i, j)$ for all $1 \leq k \leq n$.

   Parameter: Indices of nodes used as intermediate nodes.

2. Handle the base case

   $f_0(i, j) = w(v_i, v_j)$.

3. Write a recurrence for larger subproblems

   $f_{k+1}(i, j) = \min\{f_k(i, j), f_k(i, k+1) + f_k(k+1, j)\}$

Example: All-Pair Shortest Path

Floyd-Warshall algorithm is an example of a dynamic program.

Four Steps of Dynamic Programming

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   To compute distance, we compute $f_k(i, j)$ for all $1 \leq k \leq n$.

   Parameter: Indices of nodes used as intermediate nodes.

2. Handle the base case

   $f_0(i, j) = w(v_i, v_j).$

3. Write a recurrence for larger subproblems

   $f_{k+1}(i, j) = \min\{f_k(i, j), f_k(i, k+1) + f_k(k+1, j)\}$

4. Fill the table of partial solutions in a bottom-up way

   Start from f_1, then compute $f_2, f_3, \ldots, f_{n-1}$

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

A sequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

A subsequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

An increasing subsequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

A longest increasing subsequence:

5 **2** 8 6 **3** **6** **9** 7

Example: Longest Increasing Subsequence

Increasing Subsequences

Let $a[1..n]$ be an array of numbers. A subsequence of a is a sequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
An increasing subsequence is a subsequence

$$a[i_1], a[i_2], \ldots, a[i_k]$$

where $a[i_1] < a[i_2] < \ldots < a[i_k]$

example

Another longest increasing subsequence:

5 2 8 6 3 6 9 7

Example: Longest Increasing Subsequence

Question

Given a sequence $a[1..n]$ of numbers, compute a longest increasing subsequence.

Example: Longest Increasing Subsequence

Question

Given a sequence $a[1..n]$ of numbers, compute a longest increasing subsequence.

Reformulate as a Graph Question

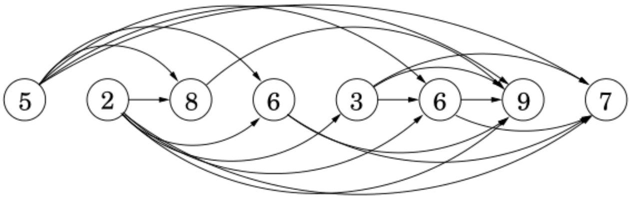
$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$

Example: Longest Increasing Subsequence

Question

Given a sequence $a[1..n]$ of numbers, compute a longest increasing subsequence.

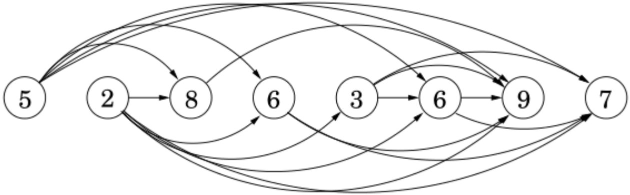
Reformulate as a Graph Question



Create an edge $(a[i], a[j])$ if $i < j$ and $a[i] < a[j]$.
Compute the longest path in this graph.

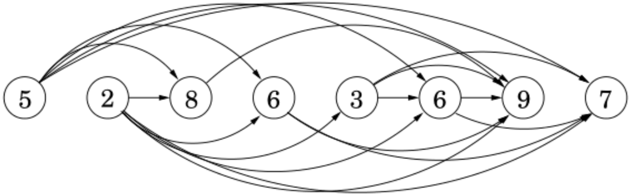
Example: Longest Increasing Subsequence

Divide into Subproblems



Example: Longest Increasing Subsequence

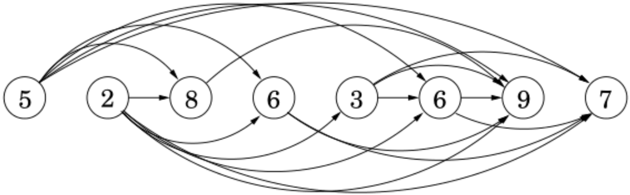
Divide into Subproblems



Let $L(i)$ denote the length of the longest path that ends at $a[i]$.

Example: Longest Increasing Subsequence

Divide into Subproblems

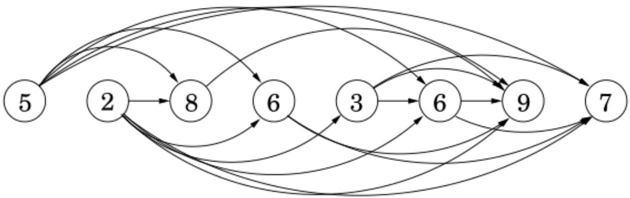


Let $L(i)$ denote the length of the longest path that ends at $a[i]$.

Then the length of the longest increasing subsequence is:

$$\max\{L(i) \mid 1 \leq i \leq n\}$$

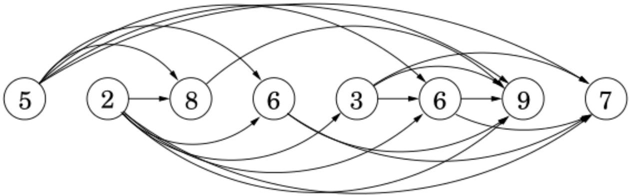
Example: Longest Increasing Subsequence



Base Case: The Smallest Subproblem

Recurrence for Larger Subproblems

Example: Longest Increasing Subsequence

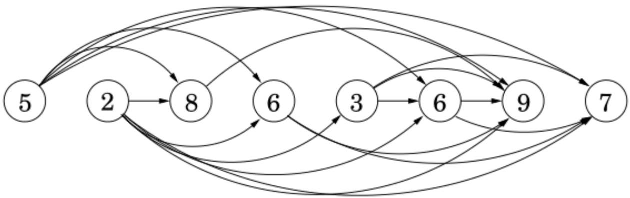


Base Case: The Smallest Subproblem

$L(1) = 1$

Recurrence for Larger Subproblems

Example: Longest Increasing Subsequence



Base Case: The Smallest Subproblem

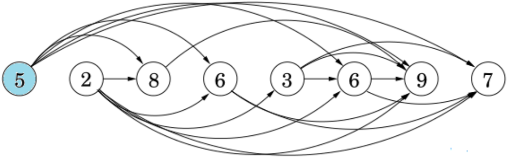
$L(1) = 1$

Recurrence for Larger Subproblems

$L(i+1) = 1$ if indegree$(a[i+1]) = 0$
$L(i+1) = 1 + \max\{L(j) \mid j < i+1, (a[j], a[i+1]) \in E\}$ otherwise

Example: Longest Increasing Subsequence

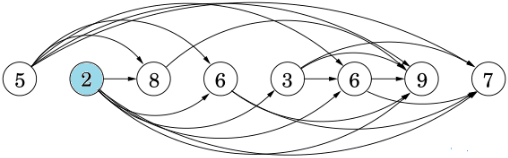
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1							

Example: Longest Increasing Subsequence

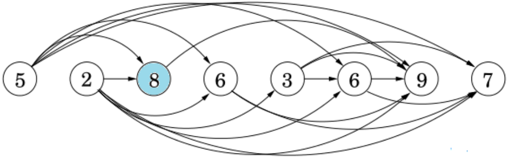
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1						

Example: Longest Increasing Subsequence

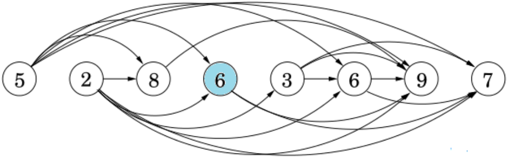
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2					

Example: Longest Increasing Subsequence

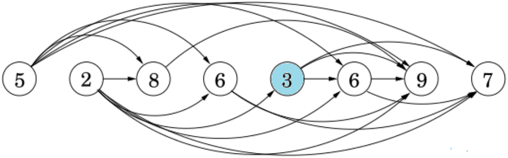
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2	2				

Example: Longest Increasing Subsequence

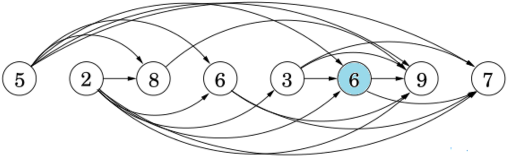
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2	2	2			

Example: Longest Increasing Subsequence

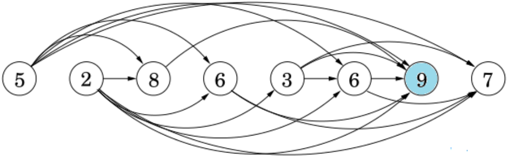
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2	2	2	3		

Example: Longest Increasing Subsequence

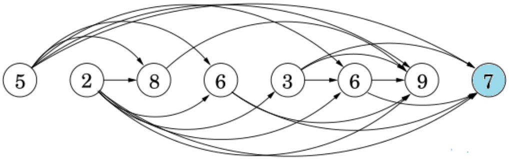
Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2	2	2	3	4	

Example: Longest Increasing Subsequence

Filling the Table



L(1)	L(2)	L(3)	L(4)	L(5)	L(6)	L(7)	L(8)
1	1	2	2	2	3	4	4

Example: Longest Increasing Subsequence

LIS($a[1..n]$)

INPUT: An integer array a of length n
OUTPUT: The length of the longest increasing subsequence in a
Create an integer array $L[1..n]$
Set every $L[i]$ to 1

```
for i = 2..n do
    for j = 1..i − 1 do
        if a[j] < a[i] then
            L[i] ← max{L[i], a[j] + 1}
Return max{L[i] | 1 ≤ i ≤ n}
```

Note: Can you extend this algorithm so that it also finds the longest increasing subsequence?

Example: Longest Increasing Subsequence

Summary

Example: Longest Increasing Subsequence

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $L(1), L(2), \ldots, L(n)$

   Parameter: The last node in the increasing subsequence

Example: Longest Increasing Subsequence

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $L(1), L(2), \ldots, L(n)$

   Parameter: The last node in the increasing subsequence

2. Handle the base case

   $L(1) = 1$

Example: Longest Increasing Subsequence

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $L(1), L(2), \ldots, L(n)$

   Parameter: The last node in the increasing subsequence

2. Handle the base case

   $L(1) = 1$

3. Write a recurrence for larger subproblems

   $L(i+1) = 1$ if $\text{indegree}(a[i+1]) = 0$

   $L(i+1) = 1 + \max\{L(j) \mid j < i+1, (a[j], a[i+1]) \in E\}$ otherwise

Example: Longest Increasing Subsequence

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $L(1), L(2), \ldots, L(n)$

   Parameter: The last node in the increasing subsequence

2. Handle the base case

   $L(1) = 1$

3. Write a recurrence for larger subproblems

   $L(i+1) = 1$ if $\text{indegree}(a[i+1]) = 0$

   $L(i+1) = 1 + \max\{L(j) \mid j < i+1, (a[j], a[i+1]) \in E\}$ otherwise

4. Fill the table of partial solutions in a bottom-up way

   Start from $L(1)$, then compute $L(2), \ldots, L(n)$

Part III: Edit Distance

Edit Distance

Motivation

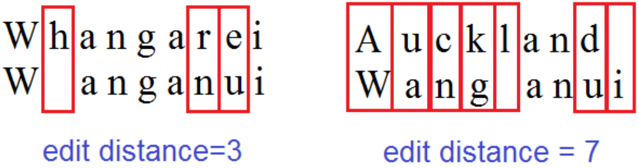
There are words that look similar and words that look different:

Whangarei Wanganui Auckland

Can we formalize this notion of similarity and disimilarity?
Can we define a distance measure on words?

Edit Distance

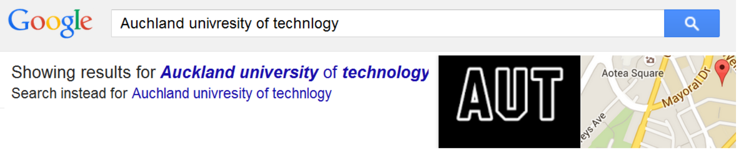
The edit distance of two words is the smallest number of edits, that are insertion, deletion, and replacement of letters, needed to transform from one word to another.



Edit Distance

Applications

- Spell Check:



- Sequence Alignment:

Align two sequences of nucleotides

AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC

Resulting alignment:

–AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC

Edit Distance

Edit Distance Problem

Given two words $a[1..m]$ and $b[1..n]$, compute the edit distance of them.

Note:

- Different ways of aligning the words result in different number of edits
- The edit distance problem asks for the best way to align the words.

S	—	N	O	W	Y
S	U	N	N	—	Y

edit distance: 3

—	S	N	O	W	—	Y
S	U	N	—	—	N	Y

edit distance: 5

Edit Distance

Observation

Suppose we would like to find the best alignment for the following:

x = Whangarei

y = Wanganui

Observation

Suppose we would like to find the best alignment for the following:
There are 3 cases:

x = Whangarei

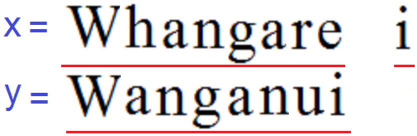
y = Wanganui

Edit Distance

Observation

Suppose we would like to find the best alignment for the following:
There are 3 cases:

Case 1: The last letter of x aligns with a blank.



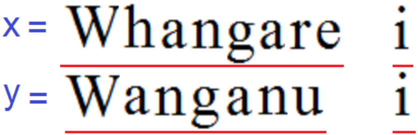
We need to then align Whangare and Wanganui

Edit Distance

Observation

Suppose we would like to find the best alignment for the following:
There are 3 cases:

Case 2: The last letter of x aligns with the last letter of y



We need to then align Whangare and Wanganu

Edit Distance

Observation

Suppose we would like to find the best alignment for the following:
There are 3 cases:

Case 3: The last letter of y aligns with a blank.

x = Whangarei
y = Wanganu i

We need to then align Whangarei and Wanganu

Edit Distance

Divide into Subproblems

Suppose we want to compute the edit distance of two words

$$x[1..m] \quad \text{and} \quad y[1..n]$$

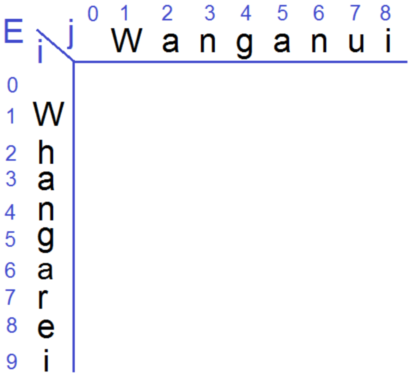
Let $E(i, j)$ be the edit distance of

$$x[1..i] \quad \text{and} \quad y[1..j]$$

We would like to find $E(m, n)$

Base Case: The Smallest Subproblem

$i = 0$ or $j = 0$



Edit Distance

Base Case: The Smallest Subproblem

$i = 0$ or $j = 0$

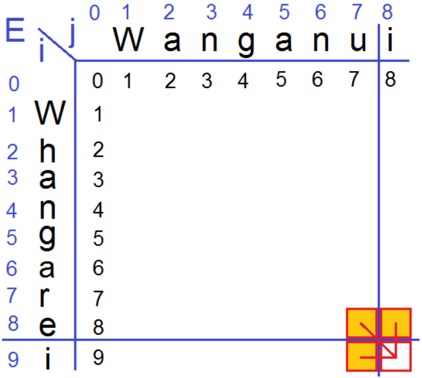
E	$i \backslash j$	0	1 W	2 a	3 n	4 g	5 a	6 n	7 u	8 i
0		0	1	2	3	4	5	6	7	8
1	W	1								
2	h	2								
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Recurrence for Larger Subproblem

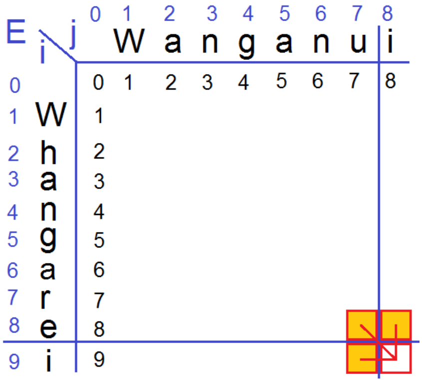
E	i \ j	0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1								
2	h	2								
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								$\boxed{}$

Recurrence for Larger Subproblem



Edit Distance

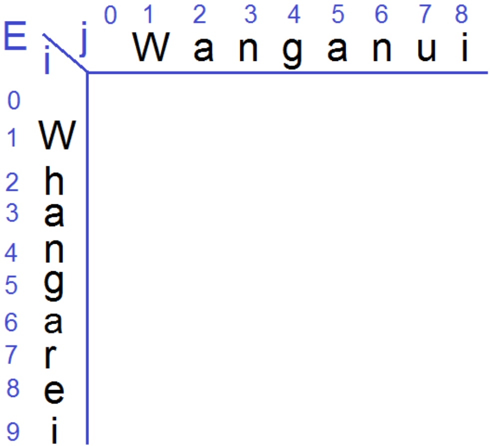
Recurrence for Larger Subproblem



$E(i+1, j+1) = \min\{E(i, j+1)+1, E[i+1, j]+1, E[i, j]+k\}$
where $k = 1$ if $a[i+1] \neq b[j+1]$, $k = 0$ if $a[i+1] = b[j+1]$.

Edit Distance

Filling the Table



Edit Distance

Filling the Table

E i \ j		0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1								
2	h	2								
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Filling the Table

E	i \ j	0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2								
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Filling the Table

E	i \ j	0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2	1	1	2	3	4	5	6	7
3	a	3								
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Filling the Table

E	i \ j	0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2	1	1	2	3	4	5	6	7
3	a	3	2	1	2	3	3	4	5	6
4	n	4								
5	g	5								
6	a	6								
7	r	7								
8	e	8								
9	i	9								

Edit Distance

Filling the Table

E	i \ j	0	1 W	2 a	3 n	4 g	5 a	6 n	7 u	8 i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2	1	1	2	3	4	5	6	7
3	a	3	2	1	2	3	3	4	5	6
4	n	4	3	2	1	2	3	3	4	5
5	g	5	4	3	2	1	2	3	4	5
6	a	6	5	4	3	2	1	2	3	4
7	r	7	6	5	4	3	2	2	3	4
8	e	8	7	6	5	4	3	3	3	4
9	i	9	8	7	6	5	4	4	4	3

Edit Distance

Filling the Table

E i \ j		0	1	2	3	4	5	6	7	8
			W	a	n	g	a	n	u	i
0		0	1	2	3	4	5	6	7	8
1	W	1	0	1	2	3	4	5	6	7
2	h	2	1	1	2	3	4	5	6	7
3	a	3	2	1	2	3	3	4	5	6
4	n	4	3	2	1	2	3	3	4	5
5	g	5	4	3	2	1	2	3	4	5
6	a	6	5	4	3	2	1	2	3	4
7	r	7	6	5	4	3	2	2	3	4
8	e	8	7	6	5	4	3	3	3	4
9	i	9	8	7	6	5	4	4	4	$\boxed{3}$

Edit Distance

EditDistance($a[1..m], b[1..n]$)

INPUT: Two words represented by two char arrays a, b
OUTPUT: The edit distance between a and b
Create an empty 2-dim array $E[1..m][1..n]$

```
for i = 0..m do
    E[i][0] ← i
for j = 0..n do
    E[0][j] ← j
for i = 1..m do
    for j = 1..n do
        k ← (a[i] ≠ b[j])
        E[i][j] ← min{E[i−1][j] + 1, E[i][j−1] + 1, E[i−1][j−1] + k}
return E[m][n]
```

Edit Distance

Summary

Edit Distance

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $E(i, j)$ for $i = 0..m$, $j = 0..n$

   Parameters: The lengths of subwords

Edit Distance

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $E(i, j)$ for $i = 0..m$, $j = 0..n$

   Parameters: The lengths of subwords

2. Handle the base case

   $E(i, 0) = i$, $E(0, j) = j$

Edit Distance

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $E(i, j)$ for $i = 0..m$, $j = 0..n$

   Parameters: The lengths of subwords

2. Handle the base case

   $E(i, 0) = i$, $E(0, j) = j$

3. Write a recurrence for larger subproblems

   $$E(i+1, j+1) = \max\{E(i+1, j) + 1, E(i, j+1) + 1, E(i, j) + k\}$$

   where $k = 0$ if $a[i+1] = b[j+1]$ and $k = 1$ otherwise.

Edit Distance

Summary

1. Parametrize the problem: Divide the problem into subproblems indexed by a parameter:

   We compute $E(i, j)$ for $i = 0..m$, $j = 0..n$

   Parameters: The lengths of subwords

2. Handle the base case

   $E(i, 0) = i$, $E(0, j) = j$

3. Write a recurrence for larger subproblems

   $$E(i+1, j+1) = \max\{E(i+1, j) + 1, E(i, j+1) + 1, E(i, j) + k\}$$

   where $k = 0$ if $a[i+1] = b[j+1]$ and $k = 1$ otherwise.

4. Fill the table of partial solutions in a bottom-up way

   Start from $E(0, j)$, then compute $E(1, j), E(2, j), \ldots, E(m, j)$