Teoria dos Grafos e Computabilidade

— Dynamic programming —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Feb 2025

Teoria dos Grafos e Computabilidade

— Dynamic programming: fundamentals —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Algorithm Design Techniques

Greedy

Build up a solution incrementally, myopically optimizing some local criterion.

Algorithm Design Techniques

Greedy

Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer

Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Algorithm Design Techniques

Greedy

Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer

Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming.

Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems

Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.

Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.
2. Greedy
   - Pro: natural approach to algorithm design.
   - Con: many greedy approaches a problem, but only some may work.
   - Con: many problems for which no greedy approach is known.

Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.
2. Greedy
    - Pro: natural approach to algorithm design.
    - Con: many greedy approaches a problem, but only some may work.
    - Con: many problems for which no greedy approach is known.
3. Divide and conquer
    - Pro: simple to develop algorithm skeleton.
    - Pro: usually reduces time for a problem known to be solvable in polynomial time.
    - Con: conquer step can be very hard to implement efficiently.

Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.
2. Greedy
   - Pro: natural approach to algorithm design.
   - Con: many greedy approaches a problem, but only some may work.
   - Con: many problems for which no greedy approach is known.
3. Divide and conquer
   - Pro: simple to develop algorithm skeleton.
   - Pro: usually reduces time for a problem known to be solvable in polynomial time.
   - Con: conquer step can be very hard to implement efficiently.
4. Dynamic programming
   - More powerful than greedy and divide-and-conquer strategies.
   - Implicitly explore space of all possible solutions.
   - Solve multiple sub-problems and build up correct solutions to larger and larger sub-problems.
   - Careful analysis needed to ensure number of sub-problems solved is polynomial in the size of the input.

History of Dynamic Programming

- Bellman pioneered the systematic study of dynamic programming in the 1950s.

History of Dynamic Programming

- Bellman pioneered the systematic study of dynamic programming in the 1950s.
- Dynamic programming = "planning over time."
- The Secretary of Defense at that time was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to" Reference:
  - Bellman, R. E., *Eye of the Hurricane, An Autobiography*.

Applications of Dynamic Programming

- Computational biology: Smith-Waterman algorithm for sequence alignment.
- Operations research: Bellman-Ford algorithm for shortest path routing in networks.
- Control theory: Viterbi algorithm for hidden Markov models.
- Computer science (theory, graphics, AI, . . .): Unix `diff` command for comparing two files.

Teoria dos Grafos e Computabilidade

— Weighted interval scheduling —

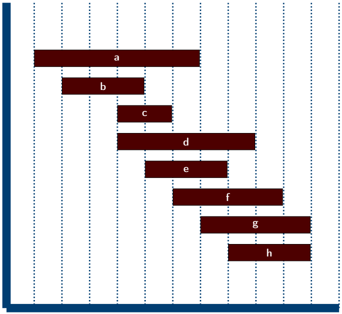
Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

INTERVAL SCHEDULING

INSTANCE Nonempty set $\{(s(i), f(i)), 1 \leq i \leq n\}$ of start and finish times of n jobs.

SOLUTION The largest subset of mutually compatible jobs.

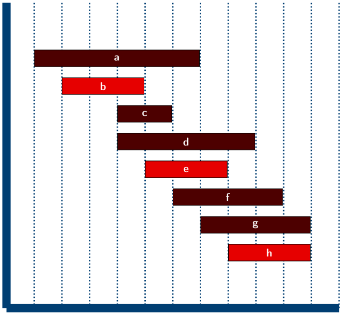


Review: Interval Scheduling

Interval Scheduling

INSTANCE Nonempty set $\{(s(i), f(i)), 1 \leq i \leq n\}$ of start and finish times of n jobs.

SOLUTION The largest subset of mutually compatible jobs.

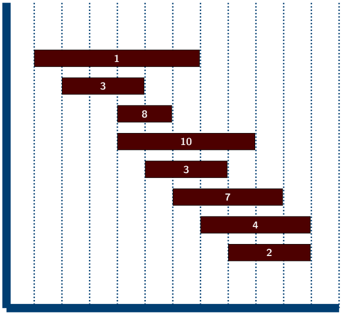


- Two jobs are compatible if they do not overlap.
- This problem models the situation where you have a resource, a set of fixed jobs, and you want to schedule as many jobs as possible.
- Greedy algorithm sort jobs in non decreasing order of finish times. Add next job to current subset only if it is compatible with previously-selected jobs.

Weighted Interval Scheduling

INSTANCE Nonempty set $\{(s_i, f_i), 1 \leq i \leq n\}$ of start and finish times of n jobs and a weight $v_i \geq 0$ associated with each job.

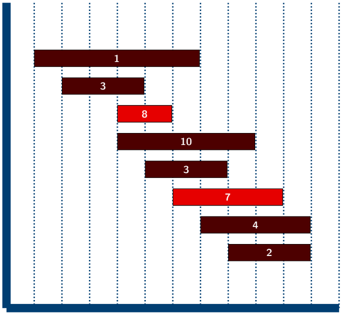
SOLUTION A set S of mutually compatible jobs such that $\sum_{i \in S} v_i$ is maximised.



WEIGHTED INTERVAL SCHEDULING

INSTANCE Nonempty set $\{(s_i, f_i), 1 \leq i \leq n\}$ of start and finish times of n jobs and a weight $v_i \geq 0$ associated with each job.

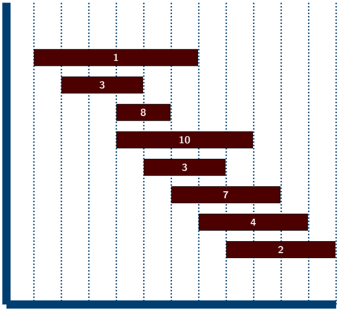
SOLUTION A set S of mutually compatible jobs such that $\sum_{i \in S} v_i$ is maximised.



- Two jobs are compatible if they do not overlap.
- This problem models the situation where you have a resource, a set of fixed jobs, and you want to schedule as many weighted jobs as possible.
- Greedy algorithm can produce arbitrarily bad results for this problem.

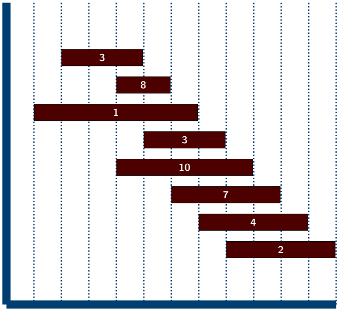
Approach

- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



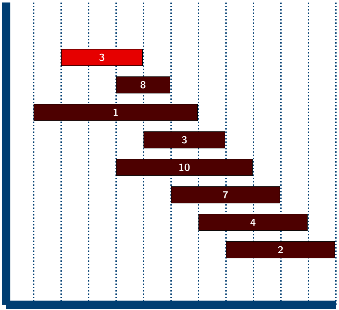
Approach

- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



Approach

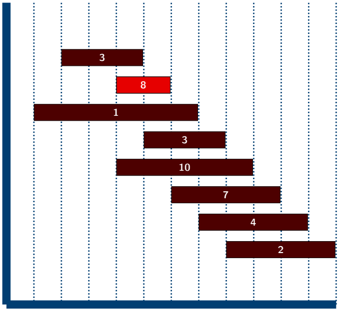
- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



- p(j) is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.
- We will develop optimal algorithm from very obvious statements about the problem.

Approach

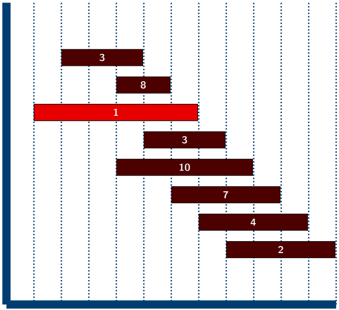
- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



- p(j) is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.
- We will develop optimal algorithm from very obvious statements about the problem.

Approach

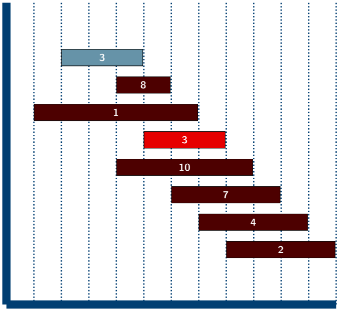
- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



- p(j) is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.
- We will develop optimal algorithm from very obvious statements about the problem.

Approach

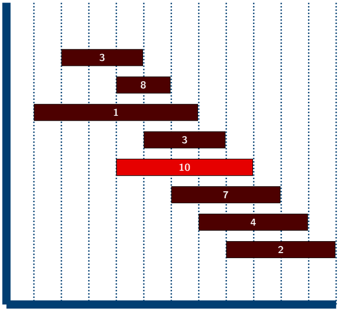
- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



- p(j) is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.
- We will develop optimal algorithm from very obvious statements about the problem.

Approach

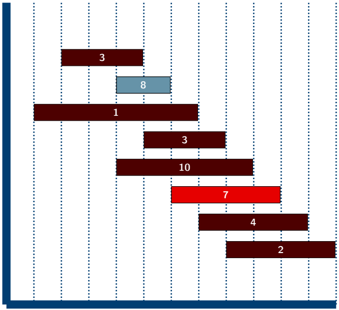
- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



- p(j) is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.
- We will develop optimal algorithm from very obvious statements about the problem.

Approach

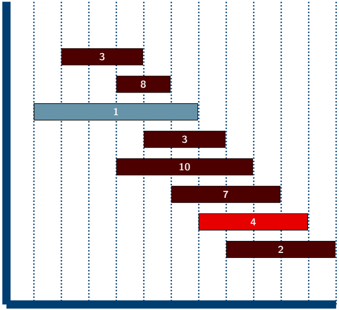
- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



- p(j) is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.
- We will develop optimal algorithm from very obvious statements about the problem.

Approach

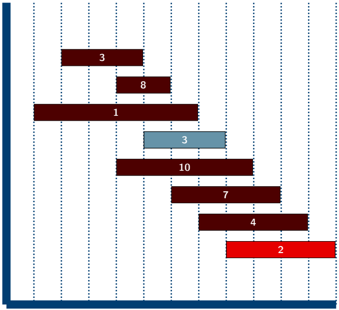
- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.



- p(j) is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.
- We will develop optimal algorithm from very obvious statements about the problem.

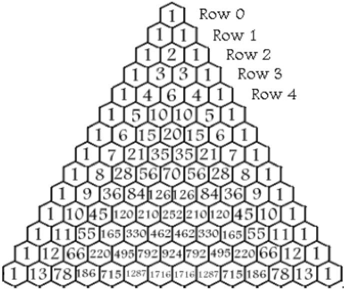
Approach

- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Request i comes before request j if $i < j$.

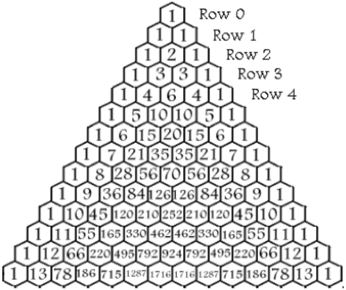


- p(j) is the largest index $i < j$ such that job i is compatible with job j. $p(j) = 0$ if there is no such job i.
- We will develop optimal algorithm from very obvious statements about the problem.

Detour: a Binomial Identity

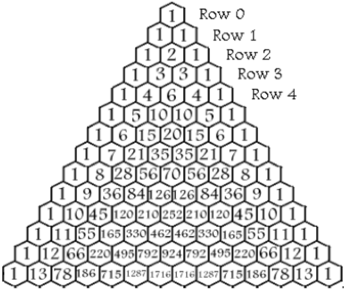


Detour: a Binomial Identity



- Pascal's triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

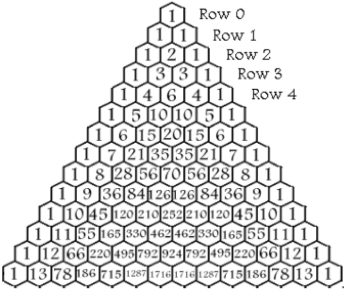
Detour: a Binomial Identity



- Pascal's triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Detour: a Binomial Identity



- Pascal's triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

- Proof: either we select the nth element or not ...

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.

  Case 1 *job n is not in $\mathcal{O}$.*

  Case 2 *job n is in $\mathcal{O}$.*

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.

  Case 1 *job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.*

  Case 2 *job n is in $\mathcal{O}$.*

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.

  Case 1 *job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.*

  Case 2 *job n is in $\mathcal{O}$.*

  - *$\mathcal{O}$ cannot use incompatible jobs $\{p(n)+1, p(n)+2, \ldots, n-1\}$.*
  - *Remaining jobs in $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, p(n)\}$.*

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.

  Case 1 *job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.*

  Case 2 *job n is in $\mathcal{O}$.*

  - *$\mathcal{O}$ cannot use incompatible jobs $\{p(n)+1, p(n)+2, \ldots, n-1\}$.*
  - *Remaining jobs in $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, p(n)\}$.*
- $\mathcal{O}$ must be the best of these two choices!

Sub-problems

- Let $\mathcal{O}$ be the optimal solution. Two cases to consider.

  Case 1 *job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.*

  Case 2 *job n is in $\mathcal{O}$.*

  - *$\mathcal{O}$ cannot use incompatible jobs $\{p(n)+1, p(n)+2, \ldots, n-1\}$.*
  - *Remaining jobs in $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, p(n)\}$.*

- $\mathcal{O}$ must be the best of these two choices!
- Suggests finding optimal solution for sub-problems consisting of jobs $\{1, 2, \ldots, j-1, j\}$, for all values of j.

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\mathsf{OPT}(j)$ be the value of this solution ($\mathsf{OPT}(0) = 0$).

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\mathrm{OPT}(j)$ be the value of this solution ($\mathrm{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\mathrm{OPT}(n)$.

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\text{OPT}(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1: $j \notin \mathcal{O}_j$:

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\text{OPT}(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1: $j \notin \mathcal{O}_j$*:* $OPT(j) = OPT(j-1)$.

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\mathrm{OPT}(j)$ be the value of this solution ($\mathrm{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\mathrm{OPT}(n)$.
- To compute $\mathrm{OPT}(j)$:

  Case 1: $j \notin \mathcal{O}_j$*:* $OPT(j) = OPT(j-1)$.

  Case 2: $j \in \mathcal{O}_j$*:*

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\text{OPT}(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1: $j \notin \mathcal{O}_j$: $OPT(j) = OPT(j-1)$.

  Case 2: $j \in \mathcal{O}_j$: $OPT(j) = v_j + OPT(p(j))$

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\text{OPT}(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1: $j \notin \mathcal{O}_j$: $OPT(j) = OPT(j-1)$.

  Case 2: $j \in \mathcal{O}_j$: $OPT(j) = v_j + OPT(p(j))$

$$\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1))$$

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\mathrm{OPT}(j)$ be the value of this solution $(\mathrm{OPT}(0) = 0)$.
- We are seeking $\mathcal{O}_n$ with a value of $\mathrm{OPT}(n)$.
- To compute $\mathrm{OPT}(j)$:

  Case 1: $j \notin \mathcal{O}_j$*:* $OPT(j) = OPT(j-1)$.

  Case 2: $j \in \mathcal{O}_j$*:* $OPT(j) = v_j + OPT(p(j))$

$$\mathrm{OPT}(j) = \max(v_j + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1))$$

- When does request j belong to $\mathcal{O}_j$?

Recursion

- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $\mathrm{OPT}(j)$ be the value of this solution ($\mathrm{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\mathrm{OPT}(n)$.
- To compute $\mathrm{OPT}(j)$:

  Case 1: $j \notin \mathcal{O}_j$: $OPT(j) = OPT(j-1)$.

  Case 2: $j \in \mathcal{O}_j$: $OPT(j) = v_j + OPT(p(j))$

$$\mathrm{OPT}(j) = \max(v_j + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1))$$

- When does request j belong to $\mathcal{O}_j$? If and only if $v_j + \mathrm{OPT}(p(j)) \geq \mathrm{OPT}(j-1)$.

Recursive Algorithm

Algorithm: Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
if j = 0 then
    return 0
else
    return max
      (v_j+Compute−opt(p(j)),Compute−opt(j-1))
end
```

- Correctness of algorithm follows by induction.
- What is the running time of the algorithm?

Recursive Algorithm

Algorithm: Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
if j = 0 then
    return 0
else
    return max
      (v_j+Compute−opt(p(j)),Compute−opt(j-1))
end
```

- Correctness of algorithm follows by induction.
- What is the running time of the algorithm? Can be exponential in n.

Recursive Algorithm

Algorithm: Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
if j = 0 then
    return 0
else
    return max
      (v_j+Compute−opt(p(j)),Compute−opt(j-1))
end
```

- Correctness of algorithm follows by induction.
- What is the running time of the algorithm? Can be exponential in n.
- When $p(j) = j - 2$, for all $j \geq 2$: recursive calls are for $j - 1$ and $j - 2$.

Example of Recursive Algorithm

$$\text{OPT}(6) =$$
$$\text{OPT}(5) =$$
$$\text{OPT}(4) =$$
$$\text{OPT}(3) =$$
$$\text{OPT}(2) =$$
$$\text{OPT}(1) =$$
$$\text{OPT}(0) = 0$$

Example of Recursive Algorithm

$$\begin{aligned}
OPT(6) &= \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) \\
OPT(5) &= \\
OPT(4) &= \\
OPT(3) &= \\
OPT(2) &= \\
OPT(1) &= \\
OPT(0) &= 0
\end{aligned}$$

Example of Recursive Algorithm

$$
\begin{aligned}
OPT(6) &= \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) \\
OPT(5) &= \max(v_5 + OPT(p(5)), OPT(4)) = \max(2 + OPT(3), OPT(4)) \\
OPT(4) &= \\
OPT(3) &= \\
OPT(2) &= \\
OPT(1) &= \\
OPT(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
OPT(6) &= \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) \\
OPT(5) &= \max(v_5 + OPT(p(5)), OPT(4)) = \max(2 + OPT(3), OPT(4)) \\
OPT(4) &= \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) \\
OPT(3) &= \\
OPT(2) &= \\
OPT(1) &= \\
OPT(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
\mathrm{OPT}(6) &= \max(v_6 + \mathrm{OPT}(p(6)), \mathrm{OPT}(5)) = \max(1 + \mathrm{OPT}(3), \mathrm{OPT}(5))\\
\mathrm{OPT}(5) &= \max(v_5 + \mathrm{OPT}(p(5)), \mathrm{OPT}(4)) = \max(2 + \mathrm{OPT}(3), \mathrm{OPT}(4))\\
\mathrm{OPT}(4) &= \max(v_4 + \mathrm{OPT}(p(4)), \mathrm{OPT}(3)) = \max(7 + \mathrm{OPT}(0), \mathrm{OPT}(3))\\
\mathrm{OPT}(3) &= \max(v_3 + \mathrm{OPT}(p(3)), \mathrm{OPT}(2)) = \max(4 + \mathrm{OPT}(1), \mathrm{OPT}(2))\\
\mathrm{OPT}(2) &=\\
\mathrm{OPT}(1) &=\\
\mathrm{OPT}(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
\mathrm{OPT}(6) &= \max(v_6 + \mathrm{OPT}(p(6)), \mathrm{OPT}(5)) = \max(1 + \mathrm{OPT}(3), \mathrm{OPT}(5))\\
\mathrm{OPT}(5) &= \max(v_5 + \mathrm{OPT}(p(5)), \mathrm{OPT}(4)) = \max(2 + \mathrm{OPT}(3), \mathrm{OPT}(4))\\
\mathrm{OPT}(4) &= \max(v_4 + \mathrm{OPT}(p(4)), \mathrm{OPT}(3)) = \max(7 + \mathrm{OPT}(0), \mathrm{OPT}(3))\\
\mathrm{OPT}(3) &= \max(v_3 + \mathrm{OPT}(p(3)), \mathrm{OPT}(2)) = \max(4 + \mathrm{OPT}(1), \mathrm{OPT}(2))\\
\mathrm{OPT}(2) &= \max(v_2 + \mathrm{OPT}(p(2)), \mathrm{OPT}(1)) = \max(4 + \mathrm{OPT}(0), \mathrm{OPT}(1))\\
\mathrm{OPT}(1) &=\\
\mathrm{OPT}(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
\mathrm{OPT}(6) &= \max(v_6 + \mathrm{OPT}(p(6)), \mathrm{OPT}(5)) = \max(1 + \mathrm{OPT}(3), \mathrm{OPT}(5))\\
\mathrm{OPT}(5) &= \max(v_5 + \mathrm{OPT}(p(5)), \mathrm{OPT}(4)) = \max(2 + \mathrm{OPT}(3), \mathrm{OPT}(4))\\
\mathrm{OPT}(4) &= \max(v_4 + \mathrm{OPT}(p(4)), \mathrm{OPT}(3)) = \max(7 + \mathrm{OPT}(0), \mathrm{OPT}(3))\\
\mathrm{OPT}(3) &= \max(v_3 + \mathrm{OPT}(p(3)), \mathrm{OPT}(2)) = \max(4 + \mathrm{OPT}(1), \mathrm{OPT}(2))\\
\mathrm{OPT}(2) &= \max(v_2 + \mathrm{OPT}(p(2)), \mathrm{OPT}(1)) = \max(4 + \mathrm{OPT}(0), \mathrm{OPT}(1))\\
\mathrm{OPT}(1) &= v_1 = 2\\
\mathrm{OPT}(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
\mathrm{OPT}(6) &= \max(v_6 + \mathrm{OPT}(p(6)), \mathrm{OPT}(5)) = \max(1 + \mathrm{OPT}(3), \mathrm{OPT}(5)) \\
\mathrm{OPT}(5) &= \max(v_5 + \mathrm{OPT}(p(5)), \mathrm{OPT}(4)) = \max(2 + \mathrm{OPT}(3), \mathrm{OPT}(4)) \\
\mathrm{OPT}(4) &= \max(v_4 + \mathrm{OPT}(p(4)), \mathrm{OPT}(3)) = \max(7 + \mathrm{OPT}(0), \mathrm{OPT}(3)) \\
\mathrm{OPT}(3) &= \max(v_3 + \mathrm{OPT}(p(3)), \mathrm{OPT}(2)) = \max(4 + \mathrm{OPT}(1), \mathrm{OPT}(2)) \\
\mathrm{OPT}(2) &= \max(v_2 + \mathrm{OPT}(p(2)), \mathrm{OPT}(1)) = \max(4 + \mathrm{OPT}(0), \mathrm{OPT}(1)) = 4 \\
\mathrm{OPT}(1) &= v_1 = 2 \\
\mathrm{OPT}(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
OPT(6) &= \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) \\
OPT(5) &= \max(v_5 + OPT(p(5)), OPT(4)) = \max(2 + OPT(3), OPT(4)) \\
OPT(4) &= \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) \\
OPT(3) &= \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6 \\
OPT(2) &= \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4 \\
OPT(1) &= v_1 = 2 \\
OPT(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
OPT(6) &= \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) \\
OPT(5) &= \max(v_5 + OPT(p(5)), OPT(4)) = \max(2 + OPT(3), OPT(4)) \\
OPT(4) &= \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7 \\
OPT(3) &= \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6 \\
OPT(2) &= \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4 \\
OPT(1) &= v_1 = 2 \\
OPT(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
OPT(6) &= \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) \\
OPT(5) &= \max(v_5 + OPT(p(5)), OPT(4)) = \max(2 + OPT(3), OPT(4)) = 8 \\
OPT(4) &= \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7 \\
OPT(3) &= \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6 \\
OPT(2) &= \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4 \\
OPT(1) &= v_1 = 2 \\
OPT(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$
\begin{aligned}
OPT(6) &= \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) = 8\\
OPT(5) &= \max(v_5 + OPT(p(5)), OPT(4)) = \max(2 + OPT(3), OPT(4)) = 8\\
OPT(4) &= \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7\\
OPT(3) &= \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6\\
OPT(2) &= \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4\\
OPT(1) &= v_1 = 2\\
OPT(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

$$\mathrm{OPT}(6) = \max(v_6 + \mathrm{OPT}(p(6)), \mathrm{OPT}(5)) = \max(1 + \mathrm{OPT}(3), \mathrm{OPT}(5)) = 8$$
$$\mathrm{OPT}(5) = \max(v_5 + \mathrm{OPT}(p(5)), \mathrm{OPT}(4)) = \max(2 + \mathrm{OPT}(3), \mathrm{OPT}(4)) = 8$$
$$\mathrm{OPT}(4) = \max(v_4 + \mathrm{OPT}(p(4)), \mathrm{OPT}(3)) = \max(7 + \mathrm{OPT}(0), \mathrm{OPT}(3)) = 7$$
$$\mathrm{OPT}(3) = \max(v_3 + \mathrm{OPT}(p(3)), \mathrm{OPT}(2)) = \max(4 + \mathrm{OPT}(1), \mathrm{OPT}(2)) = 6$$
$$\mathrm{OPT}(2) = \max(v_2 + \mathrm{OPT}(p(2)), \mathrm{OPT}(1)) = \max(4 + \mathrm{OPT}(0), \mathrm{OPT}(1)) = 4$$
$$\mathrm{OPT}(1) = v_1 = 2$$
$$\mathrm{OPT}(0) = 0$$

- Optimal solution is

Example of Recursive Algorithm

$$
\begin{aligned}
OPT(6) &= \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) = 8 \\
OPT(5) &= \max(v_5 + OPT(p(5)), OPT(4)) = \max(2 + OPT(3), OPT(4)) = 8 \\
OPT(4) &= \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7 \\
OPT(3) &= \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6 \\
OPT(2) &= \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4 \\
OPT(1) &= v_1 = 2 \\
OPT(0) &= 0
\end{aligned}
$$

- Optimal solution is job 5, job 3, and job 1.

Memoisation

- Store OPT(j) values in a cache and reuse them rather than recompute them.

Algorithm: M-Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
if j = 0 then
    return 0;
else if M[j] is not empty then
    return M[j];
else
    M[j]= max
        (v_j+M-Compute−opt(p(j)),M-Compute−opt(j-1));
    return M[j] ;
end
```

Running Time of Memoisation

Algorithm: M-Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
if j = 0 then
    return 0;
else if M[j] is not empty then
    return M[j];
else
    M[j]= max (v_j+M-Compute−opt(p(j)),M-Compute−opt(j-1));
    return M[j] ;
end
```

- Claim: running time of this algorithm is $\boxed{O(n)}$ (after sorting).

Running Time of Memoisation

Algorithm: M-Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

if $j = 0$ **then**
    return 0;
else if $M[j]$ *is not empty* **then**
    return $M[j]$;
else
    $M[j]$= max (v_j+M-Compute−opt(p(j)),M-Compute−opt(j-1));
    return $M[j]$;
end

- Claim: running time of this algorithm is $O(n)$ (after sorting).
- Time spent in a single call to `M-Compute-Opt` is $O(1)$ apart from time spent in recursive calls.

Running Time of Memoisation

Algorithm: M-Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
if j = 0 then
    return 0;
else if M[j] is not empty then
    return M[j];
else
    M[j]= max
        (v_j+M-Compute−opt(p(j)),M-Compute−opt(j-1));
    return M[j] ;
end
```

- Claim: running time of this algorithm is $O(n)$ (after sorting).
- Time spent in a single call to `M-Compute-Opt` is $O(1)$ apart from time spent in recursive calls.
- Total time spent is the order of the number of recursive calls to `M-Compute-Opt`.

Running Time of Memoisation

Algorithm: M-Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

if $j = 0$ **then**
    return 0;
else if $M[j]$ *is not empty* **then**
    return $M[j]$;
else
    $M[j]$= max (v_j+M-Compute−opt(p(j)),M-Compute−opt(j-1));
    return $M[j]$;
end

- How many such recursive calls are there in total?

Running Time of Memoisation

Algorithm: M-Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
if j = 0 then
    return 0;
else if M[j] is not empty then
    return M[j];
else
    M[j]= max
        (v_j+M-Compute−opt(p(j)),M-Compute−opt(j-1));
    return M[j] ;
end
```

- How many such recursive calls are there in total?
- Use number of filled entries in M as a measure of progress.

Running Time of Memoisation

Algorithm: M-Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.
output: A set of compatible jobs A

```
if j = 0 then
    return 0;
else if M[j] is not empty then
    return M[j];
else
    M[j]= max
        (v_j+M-Compute−opt(p(j)),M-Compute−opt(j-1));
    return M[j] ;
end
```

- How many such recursive calls are there in total?
- Use number of filled entries in M as a measure of progress.
- Each time `M-Compute-Opt` issues two recursive calls, it fills in a new entry in M.

Running Time of Memoisation

Algorithm: M-Compute-opt

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
if j = 0 then
    return 0;
else if M[j] is not empty then
    return M[j];
else
    M[j]= max
        (v_j+M-Compute−opt(p(j)),M-Compute−opt(j-1));
    return M[j] ;
end
```

- How many such recursive calls are there in total?
- Use number of filled entries in M as a measure of progress.
- Each time `M-Compute-Opt` issues two recursive calls, it fills in a new entry in M.
- Therefore, total number of recursive calls is $\boxed{O(n)}$.

From Recursion to Iteration

- Unwind the recursion and convert it into iteration.
- Can compute values in M iteratively in $O(n)$ time.
- `Find-Solution` works as before.

Algorithm: Iterative weighted interval scheduling

input : A set of weighted jobs R, index j and largest compatible indices.

output: A set of compatible jobs A

```
M[0] = 0;
foreach j ∈ [1, n] do
    M[j] = max(v_j+M[p(j)],M[j-1]);
end
```

Basic Outline of Dynamic Programming

- To solve a problem, we need a collection of sub-problems that satisfy a few properties:
  1. There are a polynomial number of sub-problems.
  2. The solution to the problem can be computed easily from the solutions to the sub-problems.
  3. There is a natural ordering of the sub-problems from "smallest" to "largest".
  4. There is an easy-to-compute recurrence that allows us to compute the solution to a sub-problem from the solutions to some smaller sub-problems.

Basic Outline of Dynamic Programming

- To solve a problem, we need a collection of sub-problems that satisfy a few properties:
  1. There are a polynomial number of sub-problems.
  2. The solution to the problem can be computed easily from the solutions to the sub-problems.
  3. There is a natural ordering of the sub-problems from "smallest" to "largest".
  4. There is an easy-to-compute recurrence that allows us to compute the solution to a sub-problem from the solutions to some smaller sub-problems.
- Difficulties in designing dynamic programming algorithms:
  1. Which sub-problems to define?
  2. How can we tie up sub-problems using a recurrence?
  3. How do we order the sub-problems (to allow iterative computation of optimal solutions to sub-problems)?

Teoria dos Grafos e Computabilidade

— Some exercises —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Feb 2025

Maximum subarray problem

The maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array $A[1...n]$ of numbers.

Maximum subarray problem

The maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array $A[1...n]$ of numbers.

Formally, the task is to find indices i and j with $1 \leq i \leq j \leq n$, such that

$$\sum_{x=i}^{j} A[x]$$

Maximum subarray problem

The maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array $A[1...n]$ of numbers.

Formally, the task is to find indices i and j with $1 \leq i \leq j \leq n$, such that

$$\sum_{x=i}^{j} A[x]$$

-2	1	-3	4	-1	2	1	-5	4

Maximum subarray problem

The maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array $A[1 \ldots n]$ of numbers.

Formally, the task is to find indices i and j with $1 \leq i \leq j \leq n$, such that

$$\sum_{x=i}^{j} A[x]$$

-2	1	-3	4	-1	2	1	-5	4

Maximum subarray problem

The maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array $A[1...n]$ of numbers.

Formally, the task is to find indices i and j with $1 \leq i \leq j \leq n$, such that

$$\sum_{x=i}^{j} A[x]$$

-2	1	-3	4	-1	2	1	-5	4

Some properties of this problem are:

- If the array contains all non-negative numbers, then the problem is trivial
- If the array contains all non-positive numbers, then a solution is any subarray of size 1;
- Several different sub-arrays may have the same maximum sum.

Maximum subarray problem

Maximum subarray problem

A>0	x	B>0

- If $x > 0$ then the answer is $\boxed{\mathrm{A} + \mathrm{x} + \mathrm{B}}$
- If $x < 0$ then the answer may be
  1. $\boxed{\max\{A, B\}}$ if $A + x < 0$
  2. $\boxed{\max\{A, B, A + x + B\}}$ if $A + x > 0$

Longest increasing subsequence

The longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.

Longest increasing subsequence

The longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.

2	3	9	5	9	8	4	0	1

Longest increasing subsequence

The **longest increasing subsequence** problem is to find a subsequence of a given sequence in which the subsequence's elements are in **sorted order**, lowest to highest, and in which the subsequence is as long as possible.

	2	3	9	5	9	8	4	0	1
L:	1	2	3	3	4	4	3	0	1

$$max\{L(1), L(2), ..., L(n)\}$$

Longest increasing subsequence

The **longest increasing subsequence** problem is to find a subsequence of a given sequence in which the subsequence's elements are in **sorted order**, lowest to highest, and in which the subsequence is as long as possible.

	2	3	9	5	9	8	4	0	1
L:	1	2	3	3	4	4	3	0	1

$$max\{L(1), L(2), ..., L(n)\}$$

Longest increasing subsequence

Placing billboards

The problem of placing billboards is defined as follows: You need to decide where to put multiple advertisement on a highway of M kms.

Placing billboards

The problem of placing billboards is defined as follows: You need to decide where to put multiple advertisement on a highway of M kms.

- There are n possible places where you can place an advertisement given by $x_1, x_2, \cdots, x_n$ in $[0, M]$.
- Placing an advertisement at x_i gives value r_i.
- You cannot put two advertisements at distance < 5kms from each other.

06	07	12	14	18	19

The problem of placing billboards is defined as follows: You need to decide where to put multiple advertisement on a highway of M kms.

- There are n possible places where you can place an advertisement given by $x_1, x_2, \cdots, x_n$ in $[0, M]$.
- Placing an advertisement at x_i gives value r_i.
- You cannot put two advertisements at distance < 5kms from each other.

	06	07	12	14	18	19
r:	05	06	05	01	02	03

Placing billboards

The problem of placing billboards is defined as follows: You need to decide where to put multiple advertisement on a highway of M kms.

- There are n possible places where you can place an advertisement given by $x_1, x_2, \cdots, x_n$ in $[0, M]$.
- Placing an advertisement at x_i gives value r_i.
- You cannot put two advertisements at distance < 5kms from each other.

	06	07	12	14	18	19
r:	05	06	05	01	02	03

Placing billboards

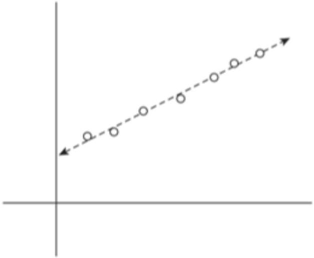
Teoria dos Grafos e Computabilidade

— Segmented Least Squares —

Silvio Jamil F. Guimarães

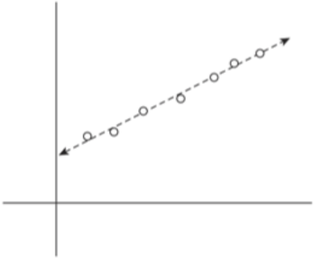
Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Least Squares Problem



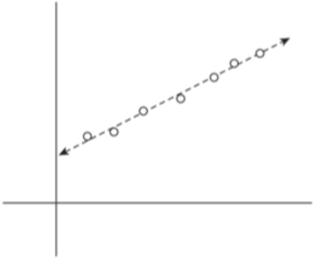
- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.

Least Squares Problem



- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.
- How do we formalise the problem?

Least Squares Problem



- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.
- How do we formalise the problem?

LEAST SQUARES

INSTANCE Set $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of n points.

SOLUTION Line $L : y = ax + b$ that minimises

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

Least Squares Problem

Least Squares

INSTANCE Set $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of n points.

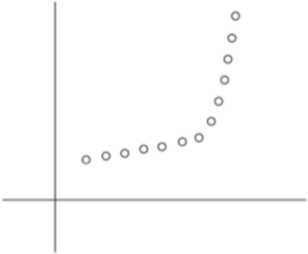
SOLUTION Line $L : y = ax + b$ that minimises

$$\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

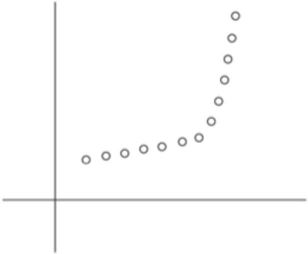
- Solution is achieved by

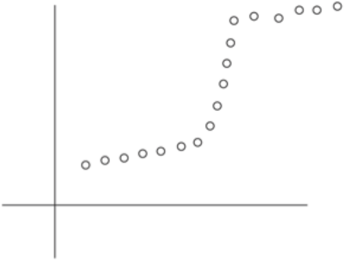
$$a = \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \text{ and } b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

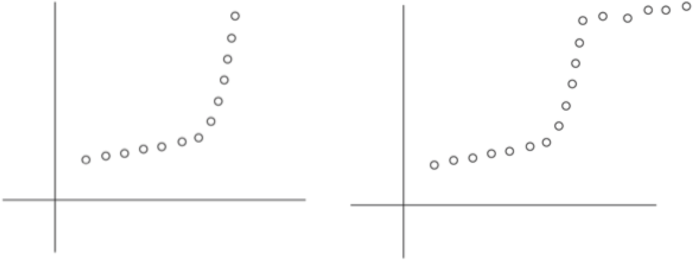
Segmented Least Squares



Segmented Least Squares

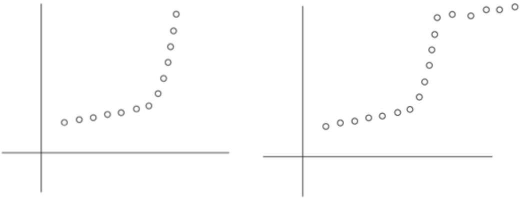






- Want to fit multiple lines through P.
- Each line must fit contiguous set of x-coordinates.
- Lines must minimise total error.

Segmented Least Squares



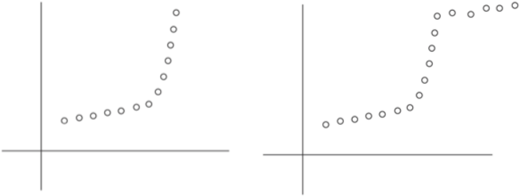
SEGMENTED LEAST SQUARES

INSTANCE Set $P = \{p_i = (x_i, y_i), 1 \leq i \leq n\}$ of n points, $x_1 < x_2 < \cdots < x_n$.

SOLUTION A integer k, a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, k lines $L_j : y = a_j x + b_j, 1 \leq j \leq k$ that minimise

$$\sum_{j=1}^{k} \text{Error}(L_j, P_j)$$

Segmented Least Squares



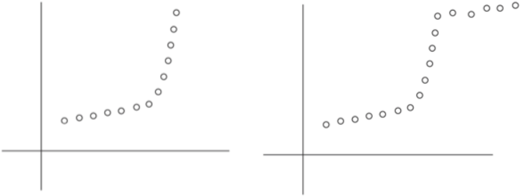
SEGMENTED LEAST SQUARES

INSTANCE Set $P = \{p_i = (x_i, y_i), 1 \leq i \leq n\}$ of n points, $x_1 < x_2 < \cdots < x_n$ and a parameter $C > 0$.

SOLUTION A integer k, a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, k lines $L_j : y = a_j x + b_j, 1 \leq j \leq k$ that minimise

$$\sum_{j=1}^{k} \text{Error}(L_j, P_j)$$

Segmented Least Squares



SEGMENTED LEAST SQUARES

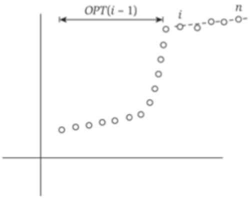
INSTANCE Set $P = \{p_i = (x_i, y_i), 1 \leq i \leq n\}$ of n points, $x_1 < x_2 < \cdots < x_n$ and a parameter $C > 0$.

SOLUTION A integer k, a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, k lines $L_j : y = a_j x + b_j, 1 \leq j \leq k$ that minimise

A subset P' of P is a **segment** if $1 \leq i < j \leq n$ exist such that $P' = \{(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_{j-1}, y_{j-1}), (x_j, y_j)\}$.

Formulating the Recursion: I

- Observation: p_n is part of some segment in the optimal solution. This segment starts at some point p_i.
- Let $\mathrm{OPT}(i)$ be the optimal value for the points $\{p_1, p_2, \ldots, p_i\}$.
- Let $e_{i,j}$ denote the minimum error of any line that fits $\{p_i, p_2, \ldots, p_j\}$.
- We want to compute $\mathrm{OPT}(n)$.



- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_n\}$, then

$$\mathrm{OPT}(n) = e_{i,n} + C + \mathrm{OPT}(i-1)$$

Formulating the Recursion: II

- Consider the sub-problem on the points $\{p_1, p_2, \ldots p_j\}$
- To obtain $\mathrm{OPT}(j)$, if the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_j\}$, then

$$\mathrm{OPT}(j) = e_{i,j} + C + \mathrm{OPT}(i-1)$$

Formulating the Recursion: II

- Consider the sub-problem on the points $\{p_1, p_2, \dots p_j\}$
- To obtain $\mathrm{OPT}(j)$, if the last segment in the optimal partition is $\{p_i, p_{i+1}, \dots, p_j\}$, then

$$\mathrm{OPT}(j) = e_{i,j} + C + \mathrm{OPT}(i-1)$$

- Since i can take only j distinct values,

$$\mathrm{OPT}(j) = \min_{1 \leq i \leq j} (e_{i,j} + C + \mathrm{OPT}(i-1))$$

- Segment $\{p_i, p_{i+1}, \dots p_j\}$ is part of the optimal solution for this sub-problem if and only if the minimum value of $\mathrm{OPT}(j)$ is obtained using index i. solution

Dynamic Programming Algorithm

$$OPT(j) = \begin{cases} 0, & \text{if } j = 0 \\ \min_{1 \leq i \leq j}(e_{ij} + c + OPT[i-1]), & \text{otherwise} \end{cases}$$

Algorithm: Segmented least squares: an iterative algorithm

input : A set of n points p_i
output: A set of compatible jobs A

```
M[0] = 0;
for j=1 to n do
    for i=1 to j do
        compute the e_ij for the segment p_i, ..., p_j;
    end
end
for j=1 to n do
    M[j] = min_{1<=i<=j}(e_ij + c + M[i-1]);
end
```

Dynamic Programming Algorithm

$$OPT(j) = \begin{cases} 0, & \text{if } j = 0 \\ \min_{1 \le i \le j}(e_{ij} + c + OPT[i-1]), & \text{otherwise} \end{cases}$$

Algorithm: Segmented least squares: an iterative algorithm

input : A set of n points p_i

output: A set of compatible jobs A

```
M[0] = 0;
for j=1 to n do
    for i=1 to j do
        compute the e_ij for the segment p_i, ..., p_j;
    end
end
for j=1 to n do
    M[j] = min_{1≤i≤j}(e_ij + c + M[i − 1]);
end
```

- Running time is $O(n^3)$, can be improved to $O(n^2)$.
- We can find the segments in the optimal solution by backtracking.

Teoria dos Grafos e Computabilidade

— Sequence alignment —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Sequence Similarity

- Given two strings, measure how similar they are.
- Given a database of strings and a query string, compute the string most similar to query in the database.
- Applications:
  - Online searches (Web, dictionary).
  - Spell-checkers.
  - Computational biology
  - Speech recognition.
  - Basis for Unix `diff`.

Defining Sequence Similarity

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

Defining Sequence Similarity

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

Defining Sequence Similarity

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

Defining Sequence Similarity

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

- Edit distance model: how many changes must you to make to one string to transform it into another?
- Changes allowed are deleting a letter, adding a letter, changing a letter.

Edit distance

INSTANCE Let two string $x = x_1x_2x_3 \ldots x_m$ and $y = y_1y_2 \ldots y_n$

SOLUTION An alignment of minimum cost.

Edit distance

INSTANCE Let two string $x = x_1x_2x_3\ldots x_m$ and $y = y_1y_2\ldots y_n$

SOLUTION An alignment of minimum cost.

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

Edit distance

INSTANCE Let two string $x = x_1x_2x_3\ldots x_m$ and $y = y_1y_2\ldots y_n$

SOLUTION An alignment of minimum cost.

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

- A matching of these sets is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \leq i \leq m$ and $1 \leq j \leq m$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.
- A matching M is an alignment if there are no "crossing pairs" in M: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.

EDIT DISTANCE

INSTANCE Let two string $x = x_1x_2x_3\ldots x_m$ and $y = y_1y_2\ldots y_n$

SOLUTION An alignment of minimum cost.

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

- A matching M is an alignment if there are no "crossing pairs" in M: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.
- The pair x_i-y_j and $x_{i'}$-yj' cross if $i < i'$', but $j - j'$.

Edit distance

INSTANCE Let two string $x = x_1x_2x_3\ldots x_m$ and $y = y_1y_2\ldots y_n$

SOLUTION An alignment of minimum cost.

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

- A matching M is an alignment if there are no "crossing pairs" in M: if $(i,j) \in M$ and $(i',j') \in M$ and $i < i'$ then $j < j'$.

$$cost(M) = \underbrace{\sum_{(x_i,y_j)\in M} \alpha_{x_iy_j}}_{\text{mismatch}} + \underbrace{\sum_{i:x_i \ \text{unmatched}} \delta + \sum_{j:x_j \ \text{unmatched}} \delta}_{\text{gaps}}$$

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- OPT(i, j) = min cost of aligning $x = x_1 \dots x_i$ and $y = y_1 \dots y_j$.
  - Case 1: OPT matches x_i-y_j so $(i, j) \in M$:

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $\mathrm{OPT}(i, j)$ = min cost of aligning $x = x_1 \ldots x_i$ and $y = y_1 \ldots y_j$.
  - Case 1: OPT matches x_i-y_j so $(i, j) \in M$:

$$\mathrm{OPT}(i, j) = \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1)$$

.

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $\text{OPT}(i, j)$ = min cost of aligning $x = x_1 \dots x_i$ and $y = y_1 \dots y_j$.
  - Case 1: OPT matches x_i-y_j so $(i, j) \in M$:

$$\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i-1, j-1)$$

.

  - Case 2a: OPT leaves x_i unmatched, so i not matched:

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $\mathrm{OPT}(i, j) =$ min cost of aligning $x = x_1 \dots x_i$ and $y = y_1 \dots y_j$.
  - Case 1: OPT matches x_i-y_j so $(i, j) \in M$:

$$\mathrm{OPT}(i, j) = \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1)$$

    .
  - Case 2a: OPT leaves x_i unmatched, so i not matched:

$$\mathrm{OPT}(i, j) = \delta + \mathrm{OPT}(i-1, j)$$

    .

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $\mathrm{OPT}(i, j) =$ min cost of aligning $x = x_1 \dots x_i$ and $y = y_1 \dots y_j$.
  - Case 1: OPT matches x_i-y_j so $(i, j) \in M$:

    $$\mathrm{OPT}(i, j) = \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1)$$

    .
  - Case 2a: OPT leaves x_i unmatched, so i not matched:

    $$\mathrm{OPT}(i, j) = \delta + \mathrm{OPT}(i-1, j)$$

    .
  - Case 2b: OPT leaves y_j unmatched, so j not matched:

    $$\mathrm{OPT}(i, j) = \delta + \mathrm{OPT}(i, j-1)$$

    .

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $\text{OPT}(i, j)$ = min cost of aligning $x = x_1 \dots x_i$ and $y = y_1 \dots y_j$.
  - Case 1: OPT matches x_i-y_j so $(i, j) \in M$:
    $$\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i-1, j-1)$$
    .
  - Case 2a: OPT leaves x_i unmatched, so i not matched:
    $$\text{OPT}(i, j) = \delta + \text{OPT}(i-1, j)$$
    .
  - Case 2b: OPT leaves y_j unmatched, so j not matched:
    $$\text{OPT}(i, j) = \delta + \text{OPT}(i, j-1)$$
    .
  - $(i, j) \in M$ if and only if minimum is achieved by the first term.
- What are the base cases?

Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- $\mathrm{OPT}(i, j) =$ min cost of aligning $x = x_1 \dots x_i$ and $y = y_1 \dots y_j$.
  - Case 1: OPT matches x_i-y_j so $(i, j) \in M$:

    $$\mathrm{OPT}(i, j) = \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1)$$

    .
  - Case 2a: OPT leaves x_i unmatched, so i not matched:

    $$\mathrm{OPT}(i, j) = \delta + \mathrm{OPT}(i-1, j)$$

    .
  - Case 2b: OPT leaves y_j unmatched, so j not matched:

    $$\mathrm{OPT}(i, j) = \delta + \mathrm{OPT}(i, j-1)$$

    .
  - $(i, j) \in M$ if and only if minimum is achieved by the first term.
- What are the base cases? $\mathrm{OPT}(i, 0) = \mathrm{OPT}(0, i) = i\delta$.

Dynamic Programming Algorithm

$$\mathrm{OPT}(i,j) = \begin{cases} j\delta, & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1), \\ \delta + \mathrm{OPT}(i-1, j), \\ \delta + \mathrm{OPT}(i, j-1) \end{cases} & \text{otherwise} \\ i\delta, & \text{if } j = 0 \end{cases}$$

- Running time is $O(mn)$. Space used in $O(mn)$.

Dynamic Programming Algorithm

$$\mathrm{OPT}(i,j) = \begin{cases} j\delta, & \textit{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1), \\ \delta + \mathrm{OPT}(i-1, j), \\ \delta + \mathrm{OPT}(i, j-1) \end{cases} & \textit{otherwise} \\ i\delta, & \textit{if } j = 0 \end{cases}$$

- Running time is $O(mn)$. Space used in $O(mn)$.
- Can compute $\mathrm{OPT}(m, n)$ in $O(mn)$ time and $O(m+n)$ space (*Hirschberg 1975*, Chapter 6.7).

Dynamic Programming Algorithm

$$
\mathrm{OPT}(i,j) = \begin{cases} j\delta, & \textit{if } \; i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1), \\ \delta + \mathrm{OPT}(i-1, j), \\ \delta + \mathrm{OPT}(i, j-1) \end{cases} & \textit{otherwise} \\ i\delta, & \textit{if } \; j = 0 \end{cases}
$$

- Running time is $O(mn)$. Space used in $O(mn)$.
- Can compute $\mathrm{OPT}(m, n)$ in $O(mn)$ time and $O(m+n)$ space (*Hirschberg 1975*, Chapter 6.7).
- Can compute *alignment* in the same bounds by combining dynamic programming with divide and conquer.

Dynamic Programming Algorithm

Longest commom subsequence

The longest commom subsequence problem is the task of finding the longest subsequence which is in two sequences x and y.

Longest commom subsequence

The longest commom subsequence problem is the task of finding the longest subsequence which is in two sequences x and y.

Formally, $w_0 w_1 \ldots w_{i-1}$ is a subsequence of $x_0 x_1 \ldots x_{m-1}$ if there exists a strictly increasing sequence of integers $(k_0, k_1, \ldots, k_{i-1})$ such that for $0 \leq k \leq i-1$. A word w is a longest common subsequence of x and y if w is a subsequence of x, a subsequence of y and its length is maximal.

Longest commom subsequence

The longest commom subsequence problem is the task of finding the longest subsequence which is in two sequences x and y.

Formally, $w_0 w_1 \ldots w_{i-1}$ is a subsequence of $x_0 x_1 \ldots x_{m-1}$ if there exists a strictly increasing sequence of integers $(k_0, k_1, \ldots, k_{i-1})$ such that for $0 \leq k \leq i-1$. A word w is a longest common subsequence of x and y if w is a subsequence of x, a subsequence of y and its length is maximal.

C	T	A	C	C	G	A	
T	A	C	A	T	T	G	T

Longest commom subsequence

The **longest commom subsequence** problem is the task of finding the longest subsequence which is in two sequences x and y.

Formally, $w_0 w_1 \ldots w_{i-1}$ is a subsequence of $x_0 x_1 \ldots x_{m-1}$ if there exists a **strictly increasing** sequence of integers $(k_0, k_1, \ldots, k_{i-1})$ such that for $0 \leq k \leq i-1$. A word w is a longest common subsequence of x and y if w is a subsequence of x, a subsequence of y and its length is maximal.

C	T	A	C	C	G	A	
T	A	C	A	T	T	G	T

Some properties of this problem are:

- the lenght of the longest subsequence must be maximal;
- may have several longest subsequences with the same size;
- it is possible to identify the subsequence by backtracking

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \begin{cases} 0, & \text{if } i = 0 \\ 1 + \text{OPT}(i-1, j-1), & \text{if } x_i = y_j \\ \max \begin{cases} \text{OPT}(i-1, j), \\ \text{OPT}(i, j-1) \end{cases} & \text{otherwise} \\ 0, & \text{if } j = 0 \end{cases}$$

	C	T	A	C	C
T					
A					
C					
A					
C					
G					

Dynamic Programming Algorithm

$$\mathrm{OPT}(i,j) = \begin{cases} 0, & \text{if } i = 0 \\ 1 + \mathrm{OPT}(i-1, j-1), & \text{if } x_i = y_j \\ \max \begin{cases} \mathrm{OPT}(i-1, j), \\ \mathrm{OPT}(i, j-1) \end{cases} & \text{otherwise} \\ 0, & \text{if } j = 0 \end{cases}$$

		C	T	A	C	C
	0	0	0	0	0	0
T						
A						
C						
A						
C						
G						

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \begin{cases} 0, & \text{if } i = 0 \\ 1 + \text{OPT}(i-1, j-1), & \text{if } x_i = y_j \\ \max \begin{cases} \text{OPT}(i-1, j), \\ \text{OPT}(i, j-1) \end{cases} & \text{otherwise} \\ 0, & \text{if } j = 0 \end{cases}$$

		C	T	A	C	C
	0	0	0	0	0	0
T	0	0	1	1	1	1
A						
C						
A						
C						
G						

Dynamic Programming Algorithm

$$\mathrm{OPT}(i,j) = \begin{cases} 0, & \textit{if } i = 0 \\ 1 + \mathrm{OPT}(i-1, j-1), & \textit{if } x_i = y_j \\ \max \begin{cases} \mathrm{OPT}(i-1, j), \\ \mathrm{OPT}(i, j-1) \end{cases} & \textit{otherwise} \\ 0, & \textit{if } j = 0 \end{cases}$$

		C	T	A	C	C
	0	0	0	0	0	0
T	0	0	1	1	1	1
A	0	0	1	2	2	2
C						
A						
C						
G						

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \begin{cases} 0, & \textit{if } \; i = 0 \\ 1 + \text{OPT}(i-1, j-1), & \textit{if } \; x_i = y_j \\ \max \begin{cases} \text{OPT}(i-1, j), \\ \text{OPT}(i, j-1) \end{cases} & \textit{otherwise} \\ 0, & \textit{if } \; j = 0 \end{cases}$$

		C	T	A	C	C
	0	0	0	0	0	0
T	0	0	1	1	1	1
A	0	0	1	2	2	2
C	0	1	1	2	3	3
A						
C						
G						

Dynamic Programming Algorithm

$$\mathrm{OPT}(i,j) = \begin{cases} 0, & \textit{if } i = 0 \\ 1 + \mathrm{OPT}(i-1, j-1), & \textit{if } x_i = y_j \\ \max \begin{cases} \mathrm{OPT}(i-1, j), \\ \mathrm{OPT}(i, j-1) \end{cases} & \textit{otherwise} \\ 0, & \textit{if } j = 0 \end{cases}$$

		C	T	A	C	C
	0	0	0	0	0	0
T	0	0	1	1	1	1
A	0	0	1	2	2	2
C	0	1	1	2	3	3
A	0	1	1	2	3	3
C						
G						

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \begin{cases} 0, & \textit{if } i = 0 \\ 1 + \text{OPT}(i-1, j-1), & \textit{if } x_i = y_j \\ \max \begin{cases} \text{OPT}(i-1, j), \\ \text{OPT}(i, j-1) \end{cases} & \textit{otherwise} \\ 0, & \textit{if } j = 0 \end{cases}$$

		C	T	A	C	C
	0	0	0	0	0	0
T	0	0	1	1	1	1
A	0	0	1	2	2	2
C	0	1	1	2	3	3
A	0	1	1	2	3	3
C	0	1	1	2	3	4
G						

Dynamic Programming Algorithm

$$\mathrm{OPT}(i,j) = \begin{cases} 0, & \textit{if } \ i = 0 \\ 1 + \mathrm{OPT}(i-1, j-1), & \textit{if } \ x_i = y_j \\ \max \begin{cases} \mathrm{OPT}(i-1, j), \\ \mathrm{OPT}(i, j-1) \end{cases} & \textit{otherwise} \\ 0, & \textit{if } \ j = 0 \end{cases}$$

		C	T	A	C	C
	0	0	0	0	0	0
T	0	0	1	1	1	1
A	0	0	1	2	2	2
C	0	1	1	2	3	3
A	0	1	1	2	3	3
C	0	1	1	2	3	4
G	0	1	1	2	3	4

Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \begin{cases} 0, & \textit{if } \ i = 0 \\ 1 + \text{OPT}(i-1, j-1), & \textit{if } \ x_i = y_j \\ \max \begin{cases} \text{OPT}(i-1, j), \\ \text{OPT}(i, j-1) \end{cases} & \textit{otherwise} \\ 0, & \textit{if } \ j = 0 \end{cases}$$

		C	T	A	C	C
	0	0	0	0	0	0
T	0	0	1	1	1	1
A	0	0	1	2	2	2
C	0	1	1	2	3	3
A	0	1	1	2	3	3
C	0	1	1	2	3	4
G	0	1	1	2	3	4

C	T	A	C	C	

T	A	C	A	C	G

Longest palindrome

The longest palindrome problem is the task of finding the longest subsequence which is a palindrome.

Longest palindrome

The **longest palindrome** problem is the task of finding the longest subsequence which is a palindrome.

Formally, $w_0 w_1 \ldots w_{i-1}$ is a subsequence of $x_0 x_1 \ldots x_{m-1}$ and w is a palindrome. A word w is a subsequence of x its length is maximal.

Longest palindrome

The **longest palindrome** problem is the task of finding the longest subsequence which is a palindrome.

Formally, $w_0 w_1 \ldots w_{i-1}$ is a subsequence of $x_0 x_1 \ldots x_{m-1}$ and w is a palindrome. A word w is a subsequence of x its length is maximal.

C	T	A	T	C	G	T	C	A	T

Longest palindrome

The **longest palindrome** problem is the task of finding the longest subsequence which is a palindrome.

Formally, $w_0 w_1 \dots w_{i-1}$ is a subsequence of $x_0 x_1 \dots x_{m-1}$ and w is a palindrome. A word w is a subsequence of x its length is maximal.

C	T	A	T	C	G	T	C	A	T

How to find the size of the longest palindrome?

Longest palindrome

The **longest palindrome** problem is the task of finding the longest subsequence which is a palindrome.

Formally, $w_0 w_1 \ldots w_{i-1}$ is a subsequence of $x_0 x_1 \ldots x_{m-1}$ and w is a palindrome. A word w is a subsequence of x its length is maximal.

C	T	A	T	C	G	T	C	A	T

How to find the size of the longest palindrome?

Longest increasing subsequence

The **longest increasing subsequence** –LIS– problem is to find a subsequence of a given sequence in which the subsequence's elements are in **sorted order**, lowest to highest, and in which the subsequence is as long as possible.

Longest increasing subsequence

The longest increasing subsequence –LIS– problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.

2	3	9	5	9	8	4	0	1

Longest increasing subsequence

The longest increasing subsequence –LIS– problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.

	2	3	9	5	9	8	4	0	1
L:	1	2	3	3	4	4	3	0	1

$$max\{L(1), L(2), ..., L(n)\}$$

Longest increasing subsequence

The **longest increasing subsequence** –LIS– problem is to find a subsequence of a given sequence in which the subsequence's elements are in **sorted order**, lowest to highest, and in which the subsequence is as long as possible.

2	3	9	5	9	8	4	0	1
1	2	3	3	4	4	3	0	1

$$max\{L(1), L(2), ..., L(n)\}$$

The longest increasing subsequence –LIS– problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.

2	3	9	5	9	8	4	0	1
1	2	3	3	4	4	3	0	1

How to find the size of the LIS by using another strategy?

Longest increasing subsequence

The longest increasing subsequence –LIS– problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.

2	3	9	5	9	8	4	0	1

1 2 3 3 4 4 3 0 1

How to find the size of the LIS by using another strategy?

Teoria dos Grafos e Computabilidade

— Shortest Path Problem —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Feb 2025

Shortest Path Problem

- $G = (V, E)$ is a connected directed graph. Each edge e has a length $l_e \geq 0$.
- V has n nodes and E has m edges.
- Length of a path P is the sum of lengths of the edges in P.
- Goal is to determine the shortest path from some start node s to each node in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

Shortest Path Problem

- $G = (V, E)$ is a connected directed graph. Each edge e has a length $l_e \geq 0$.
- V has n nodes and E has m edges.
- Length of a path P is the sum of lengths of the edges in P.
- Goal is to determine the shortest path from some start node s to each node in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

Shortest Paths

INSTANCE A directed graph $G(V, E)$, a function $l : E \rightarrow \mathbb{R}^+$, and a node $s \in V$

SOLUTION A set $\{P_u, u \in V\}$, where P_u is the shortest path in G from s to u.

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

Algorithm: Shortes path algorithm – Dijkstra)

---

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;
while $S \neq V$ **do**
  Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
  Add v to S and define d[v] = d'[v];
end

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

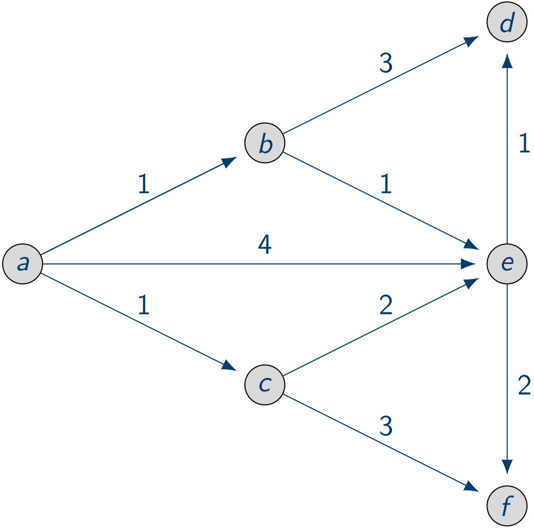
Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

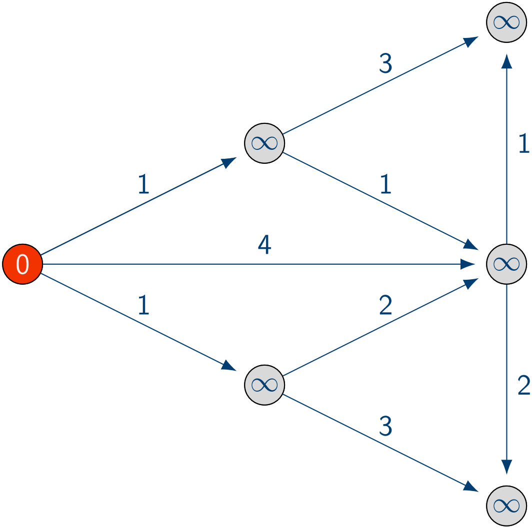
Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;
while $S \neq V$ **do**
    Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
    Add v to S and define d[v] = d'[v];
end

- Can modify algorithm to compute the shortest paths themselves: record the predecessor u that minimises $d'(v)$.

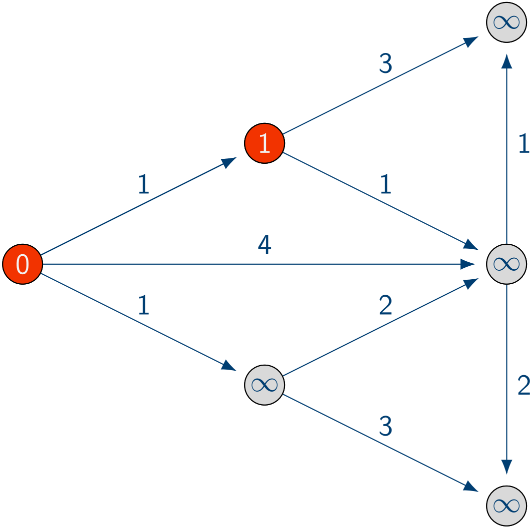
Example of Dijkstra's Algorithm



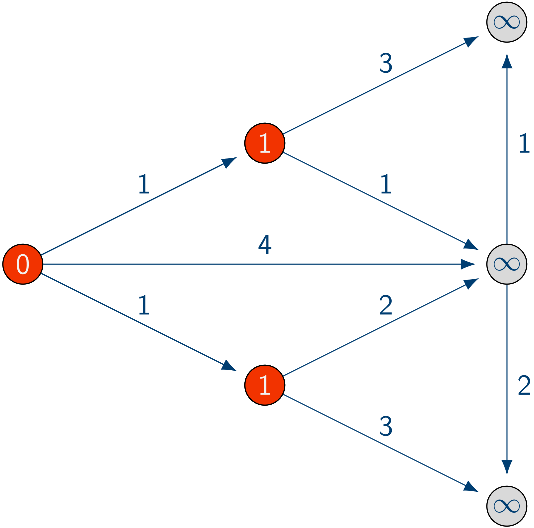
Example of Dijkstra's Algorithm



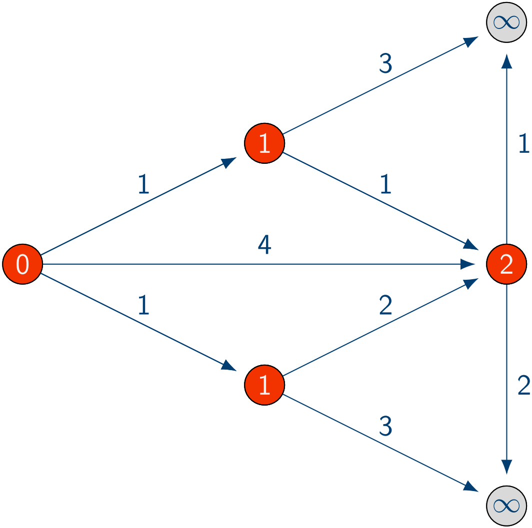
Example of Dijkstra's Algorithm



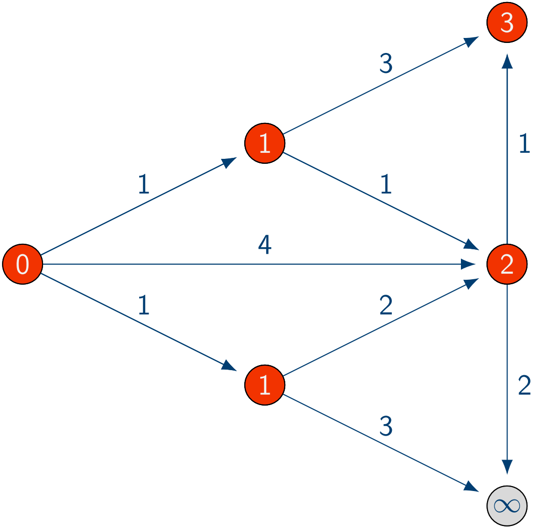
Example of Dijkstra's Algorithm



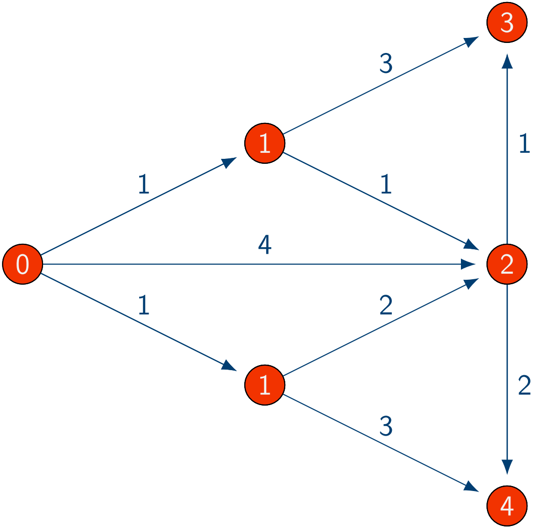
Example of Dijkstra's Algorithm



Example of Dijkstra's Algorithm



Example of Dijkstra's Algorithm

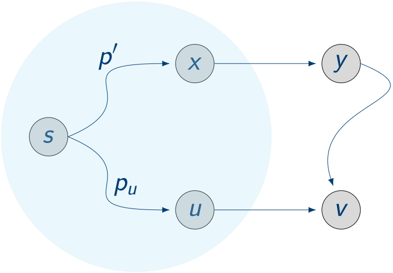


Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive step: we add the node v to S. Let u be the v's predecessor on the path P_v. Could there be a shorter path P from s to v?

Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive step: we add the node v to S. Let u be the v's predecessor on the path P_v. Could there be a shorter path P from s to v?



The alternate $s - v$ path P through x and y already too long by the time it had left the set S

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths.
- Union of shortest paths output form a tree. Why?

Implementing Dijkstra's Algorithm

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;

while $S \neq V$ **do**
    Select a node $v \notin S$ with at least one edge from S for which
        $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
    Add v to S and define d[v] = d'[v];
end

Implementing Dijkstra's Algorithm

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- How many iterations are there of the while loop? .

Implementing Dijkstra's Algorithm

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;
while $S \neq V$ **do**
    Select a node $v \notin S$ with at least one edge from S for which
        $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
    Add v to S and define d[v] = d'[v];
end

- How many iterations are there of the while loop? $n - 1$.

Implementing Dijkstra's Algorithm

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- How many iterations are there of the while loop? $n - 1$.
- In each iteration, for each node $v \notin S$, compute

$$\min_{e=(u,v),u\in S} d(u) + l_e.$$

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;

while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;
while $S \neq V$ **do**
    Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
    Add v to S and define d[v] = d'[v];
end

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a priority queue.
- Determine the next node v to add to S using ExtractMin.
- After adding v, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$, update w's key using ChangeKey.

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;

while $S \neq V$ **do**
  Select a node $v \notin S$ with at least one edge from S for which
    $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
  Add v to S and define d[v] = d'[v];
end

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a priority queue.
- Determine the next node v to add to S using ExtractMin.
- After adding v, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$, update w's key using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked?

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortes path algorithm – Dijkstra)

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;
while $S \neq V$ **do**
    Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
    Add v to S and define d[v] = d'[v];
end

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a priority queue.
- Determine the next node v to add to S using ExtractMin.
- After adding v, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$, update w's key using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked? $n - 1$ and m times, respectively.

Single Source Shortest Path Problem

- $G = (V, E)$ is a connected directed graph. Each edge e has a length l_e. Note that the weights may be negative.
- V has n nodes and E has m edges.
- Length of a path P is the sum of lengths of the edges in P.
- Goal is to determine the shortest path from some start node s to all other nodes in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

Single Source Shortest Path Problem

- $G = (V, E)$ is a connected directed graph. Each edge e has a length l_e. Note that the weights may be negative.
- V has n nodes and E has m edges.
- Length of a path P is the sum of lengths of the edges in P.
- Goal is to determine the shortest path from some start node s to all other nodes in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

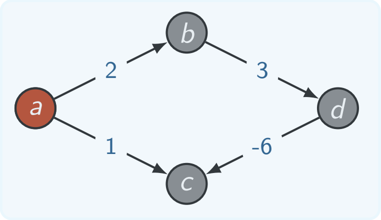
SHORTEST PATHS

INSTANCE A directed graph $G(V, E)$, a function $l : E \rightarrow \mathbb{R}$, and a node $s \in V$

SOLUTION A set $\{P_u, u \in V\}$, where P_u is the shortest path in G from s to u.

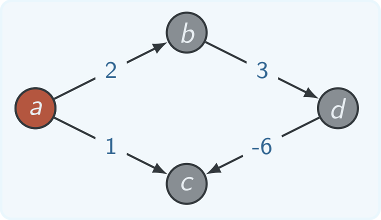
Bellman-Ford Algorithm

Dijkstra – Can fail if negative edge costs.

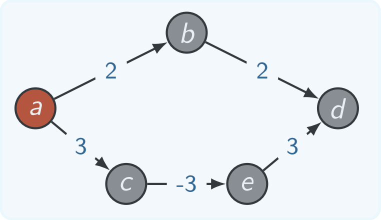


Bellman-Ford Algorithm

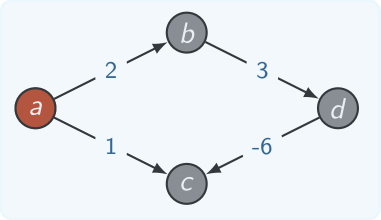
Dijkstra – Can **fail** if negative edge costs.



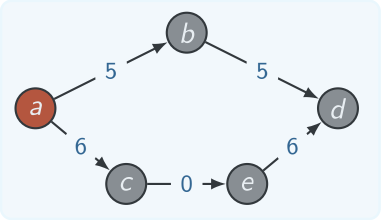
Re-weighting – Adding a **constant** to every edge weight can fail



Dijkstra – Can fail if negative edge costs.

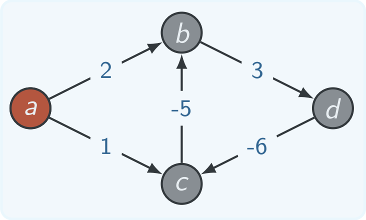


Re-weighting – Adding a constant to every edge weight can fail



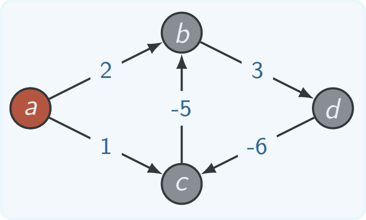
Bellman-Ford Algorithm

If some path from s to t contains a negative cost cycle, there does not exist a shortest s-t path; otherwise, there exists one that is simple.



Bellman-Ford Algorithm

If some path from s to t contains a negative cost cycle, there does not exist a shortest s-t path; otherwise, there exists one that is simple.



The Bellman-Ford algorithm is a way to find single source shortest paths in a graph with negative edge weights (but no negative cycles).

OPT(i, v) = length of shortest v-t path P using at most i edges.

Bellman-Ford Algorithm

OPT(i, v) = length of shortest v-t path P using at most i edges.

- Case 1: P uses at most $i-1$ edges.

$$OPT(i, v) = OPT(i-1, v)$$

Bellman-Ford Algorithm

OPT(i, v) = length of shortest v-t path P using at most i edges.

- Case 1: P uses at most $i-1$ edges.

$$OPT(i, v) = OPT(i-1, v)$$

- Case 2: P uses exactly i edges
  - if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most $i-1$ edges

Bellman-Ford Algorithm

OPT(i, v) = length of shortest v-t path P using at most i edges.

- Case 1: P uses at most $i-1$ edges.

$$OPT(i, v) = OPT(i-1, v)$$

- Case 2: P uses exactly i edges
  - if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most $i-1$ edges

$$OPT(i, v) = \begin{cases} 0, & \text{if } i = 0 \\ \min\begin{cases} OPT(i-1, v) \\ \min\{OPT(i-1, w) + c_{vw}\} \end{cases}, & \text{otherwise} \end{cases}$$

Algorithm: Shortest path algorithm – Bellman-Ford

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
foreach v ∈ V do d[0, u] = ∞;
Initially d[0, s] = 0;

for i = 1 to n − 1 do
    foreach v ∈ V do
        d[i, v] = d[i − 1, v]
    end
    foreach edge (v, w) ∈ E do
        d[i, v] = min{d[i, v], d[i − 1, w] + c_vw}
    end
end
```

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Bellman-Ford

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

foreach $v \in V$ **do** $d[0, u] = \infty$;
Initially $d[0, s] = 0$;

for $i = 1$ *to* $n - 1$ **do**
  foreach $v \in V$ **do**
    $d[i, v] = d[i - 1, v]$
  end
  foreach *edge* $(v, w) \in E$ **do**
    $d[i, v] = \min\{d[i, v], d[i - 1, w] + c_{vw}\}$
  end
end

- Computational cost: O(mn)

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Bellman-Ford

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

foreach $v \in V$ **do** $d[0, u] = \infty$;
Initially $d[0, s] = 0$;

for $i = 1$ *to* $n - 1$ **do**
  foreach $v \in V$ **do**
    $d[i, v] = d[i - 1, v]$
  end
  foreach *edge* $(v, w) \in E$ **do**
    $d[i, v] = \min\{d[i, v], d[i - 1, w] + c_{vw}\}$
  end
end

- Computational cost: O(mn)
- For finding the shortest paths, it is necessary to maintain a successor for each table entry.

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Bellman-Ford

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

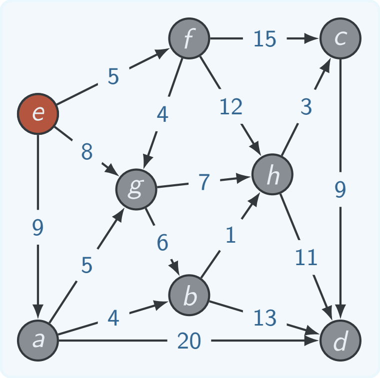
foreach $v \in V$ **do** $d[0, u] = \infty$;
Initially $d[0, s] = 0$;

for $i = 1$ *to* $n - 1$ **do**
    foreach $v \in V$ **do**
        $d[i, v] = d[i - 1, v]$
    end
    foreach *edge* $(v, w) \in E$ **do**
        $d[i, v] = \min\{d[i, v], d[i - 1, w] + c_{vw}\}$
    end
end

- Computational cost: O(mn)
- For finding the shortest paths, it is necessary to maintain a successor for each table entry.

How to detect negative cycles?

Shortest path – an example



Compute the shortest path from e to all other nodes!