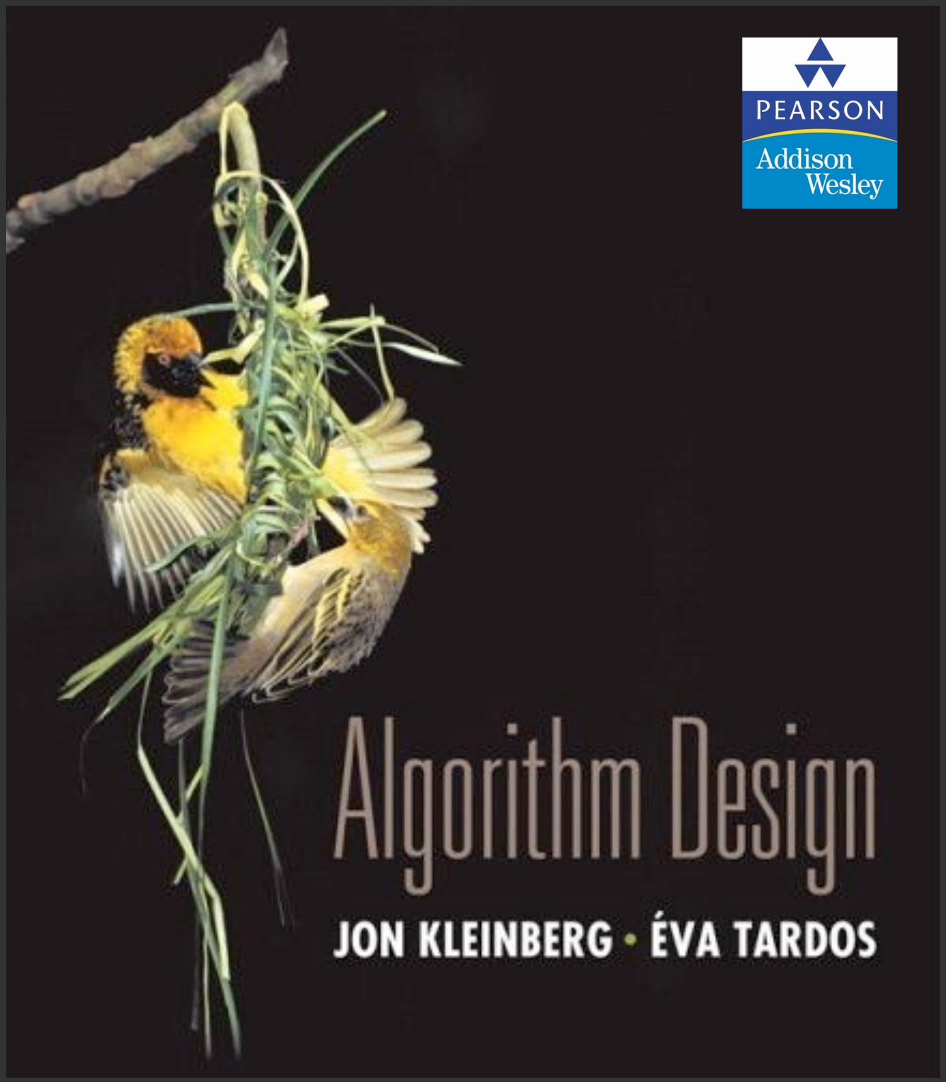


6. Dynamic Programming I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 2/10/16 9:26 AM

Algorithmic paradigms

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

Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

fancy name for
caching away intermediate results
in a table for later reuse

Dynamic programming history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

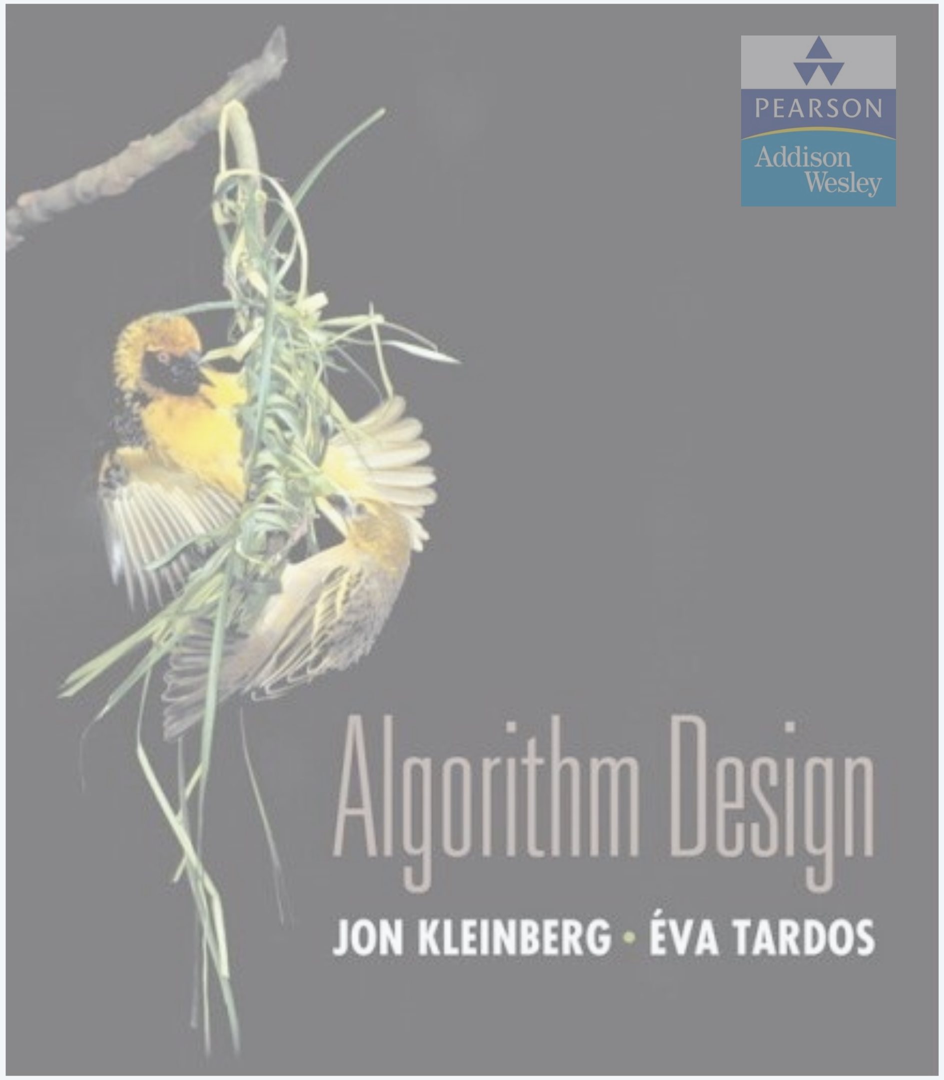
Dynamic programming applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ...
- ...

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context-free grammars.
- ...



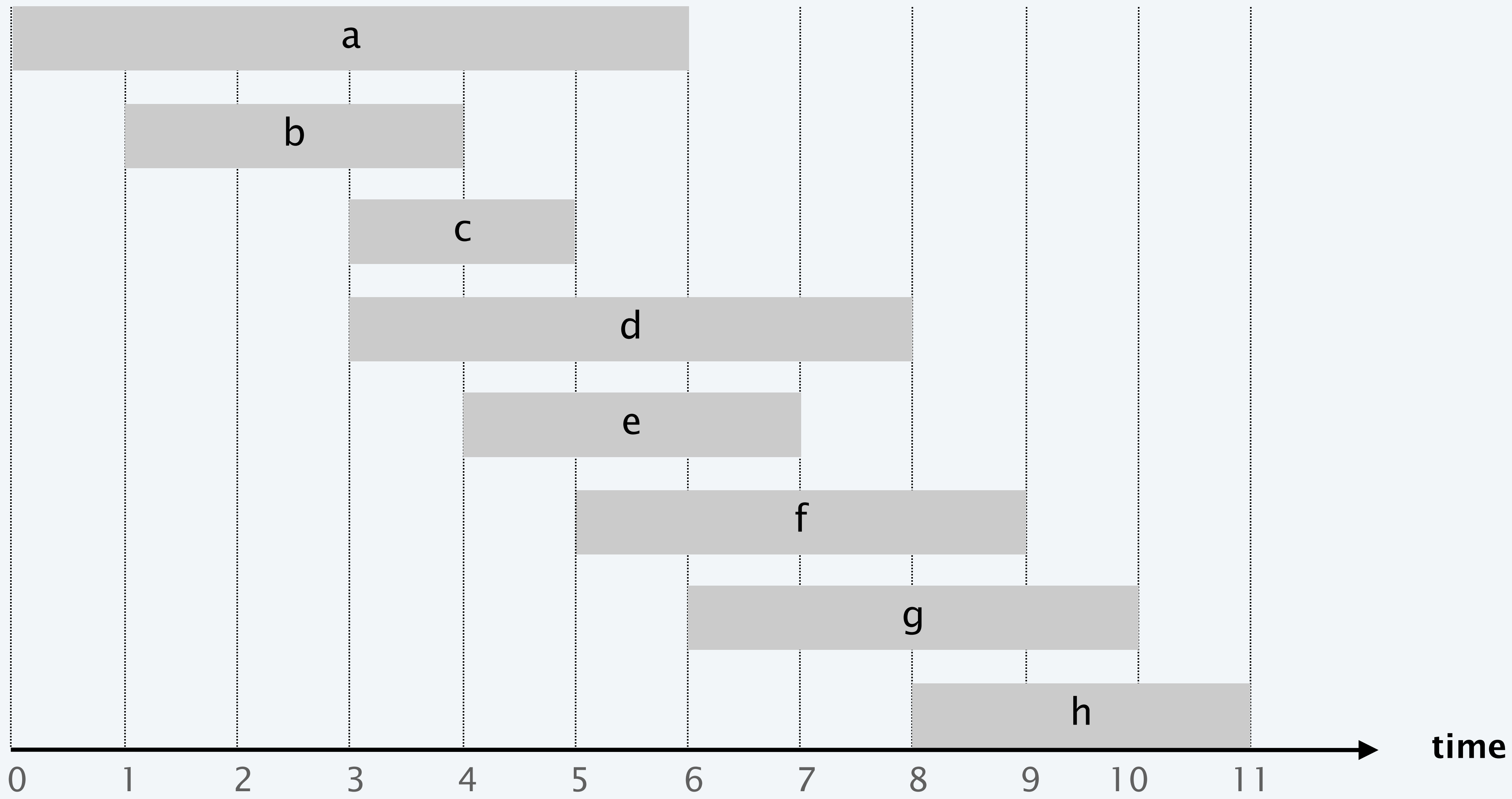
6. DYNAMIC PROGRAMMING I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Weighted interval scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



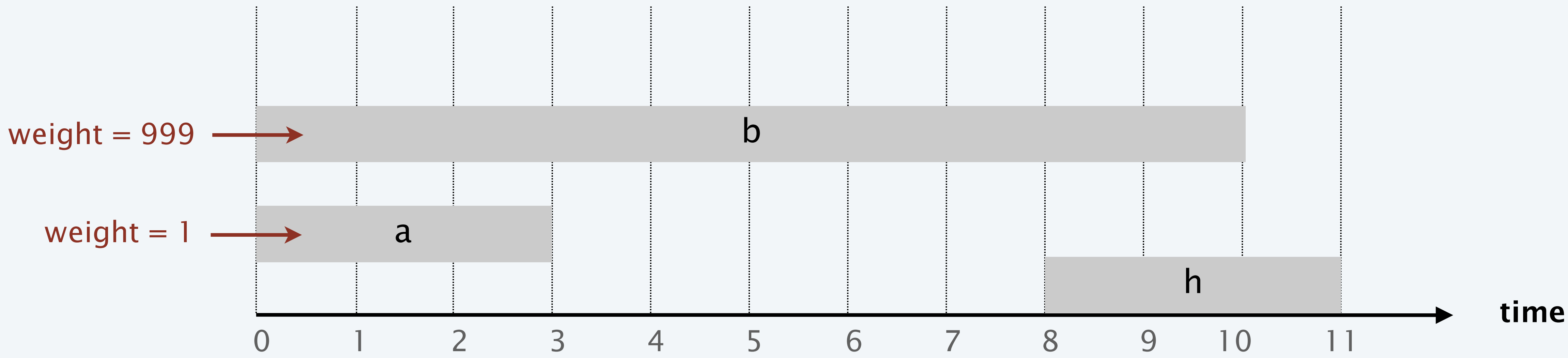
Earliest-finish-time first algorithm

Earliest finish-time first.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.

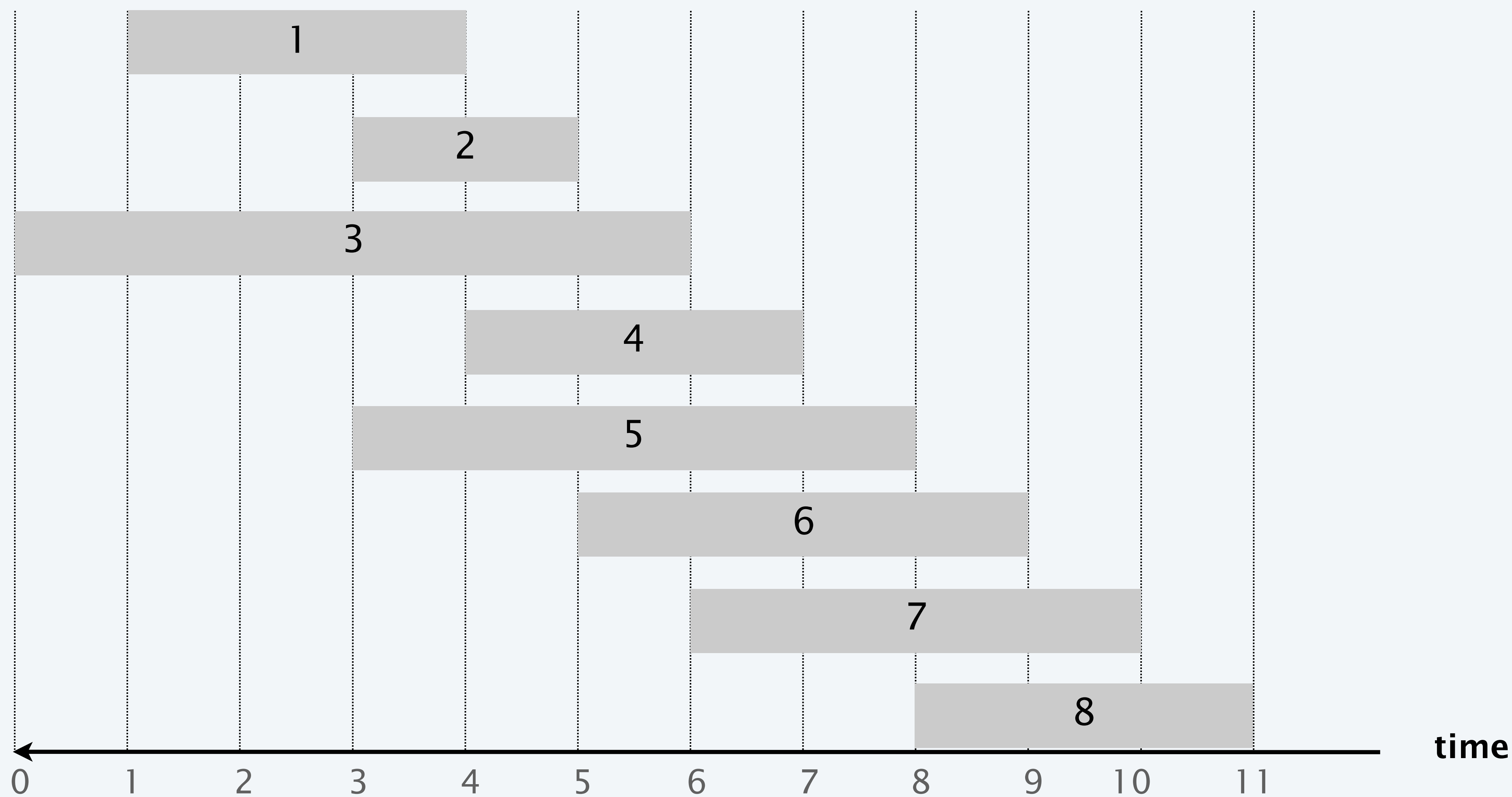


Weighted interval scheduling

Notation. Label jobs by finishing time: $f_1 \le f_2 \le \ldots \le f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex. $p(8) = 5, p(7) = 3, p(2) = 0.$



Dynamic programming: binary choice

Notation. $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, ..., j$.

Case 1. OPT selects job j.

- Collect profit v_j.
- Can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, ..., j - 1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., p(j)$.

optimal substructure property
(proof via exchange argument)

Case 2. OPT does not select job j.

- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., j - 1$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Weighted interval scheduling: brute force

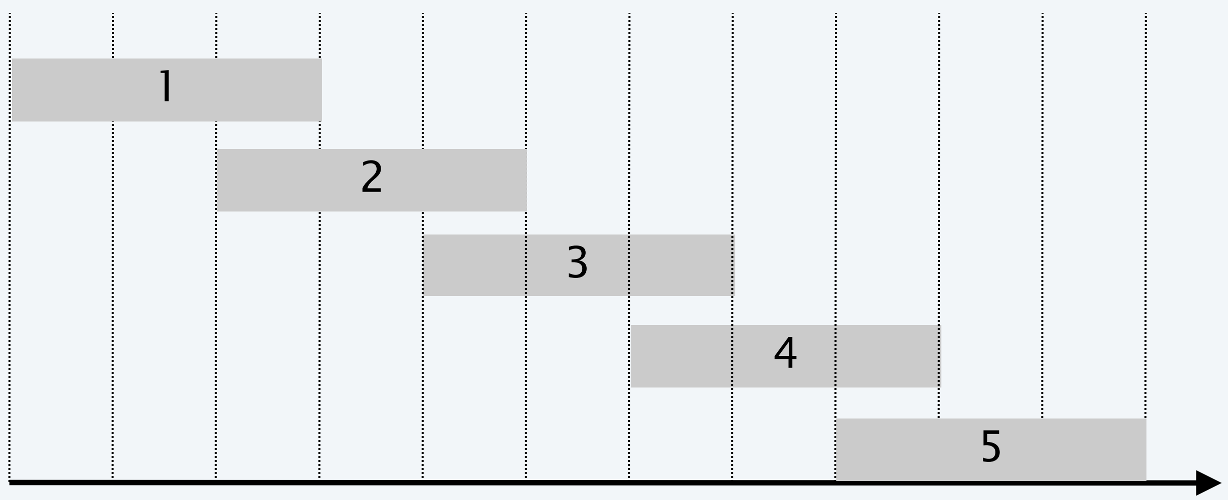
```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

Compute-Opt(j)
if j = 0
   return 0.
else
   return max(v[j] + Compute-Opt(p[j]), Compute-Opt(j–1)).
```

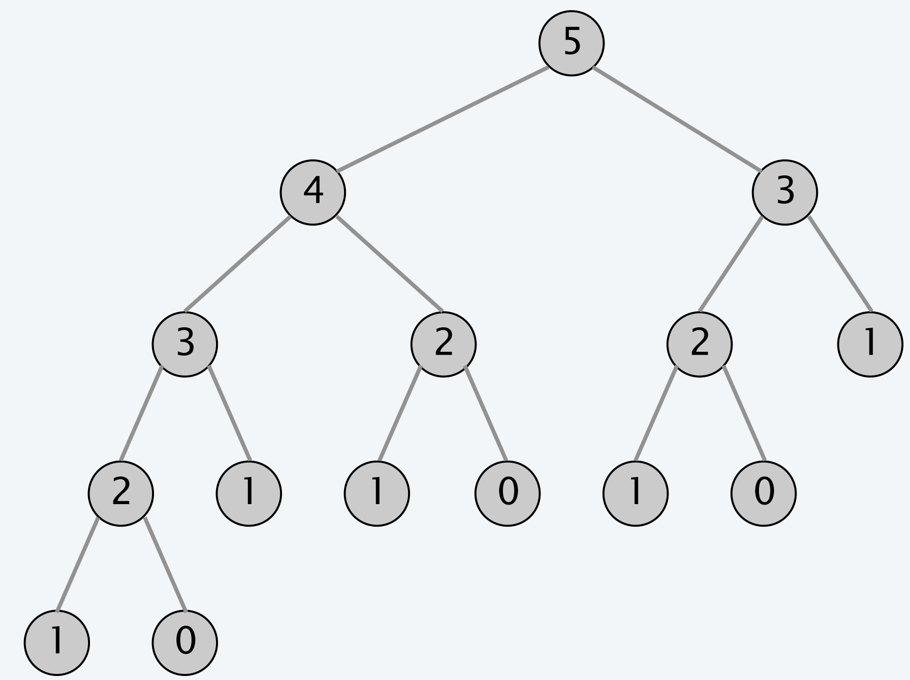
Weighted interval scheduling: brute force

Observation. Recursive algorithm fails spectacularly because of redundant subproblems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j–2



recursion tree

Weighted interval scheduling: memoization

Memoization. Cache results of each subproblem; lookup as needed.

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
________________
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j – 1)).
return M[j].
```

Weighted interval scheduling: running time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- $\text{M-COMPUTE-OPT}(j)$: each invocation takes $O(1)$ time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure $\Phi = \#$ nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of $\text{M-COMPUTE-OPT}(n)$ is $O(n)$. ▪

Remark. $O(n)$ if jobs are presorted by start and finish times.

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass.

```
Find-Solution(j)
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j-1])
   return { j } ∪ Find-Solution(p[j]).
else
   return Find-Solution(j-1).
```

Analysis. # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted interval scheduling: bottom-up

Bottom-up dynamic programming. Unwind recursion.

BOTTOM-UP $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

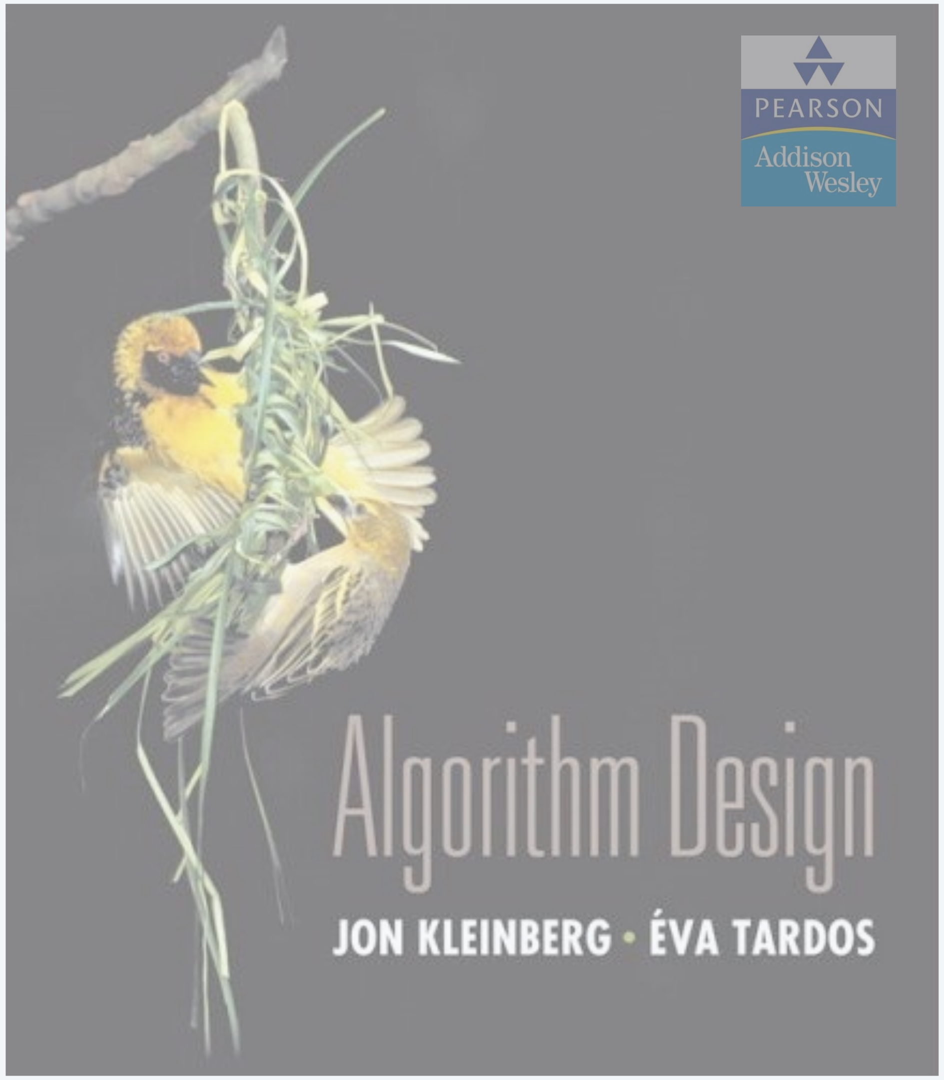
Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$.

$M[0] \leftarrow 0$.

FOR j = 1 TO n

  $M[j] \leftarrow \max \{ v_j + M[p(j)],\ M[j-1] \}$.



6. Dynamic Programming I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Knapsack problem

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance (weight limit W = 11)

Ex. $\{ 1, 2, 5 \}$ has value 35.

Ex. $\{ 3, 4 \}$ has value 40.

Ex. $\{ 3, 5 \}$ has value 46 (but exceeds weight limit).

Greedy by value. Repeatedly add item with maximum v_i.

Greedy by weight. Repeatedly add item with minimum w_i.

Greedy by ratio. Repeatedly add item with maximum ratio v_i / w_i.

Observation. None of greedy algorithms is optimal.

Dynamic programming: false start

Def. $OPT(i)$ = max profit subset of items $1, \ldots, i$.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$.

optimal substructure property
(proof via exchange argument)

Case 2. OPT selects item i.

- Selecting item i does not immediately imply that we will have to reject other items.
- Without knowing what other items were selected before i, we don't even know if we have enough room for i.

Conclusion. Need more subproblems!

Dynamic programming: adding a new variable

Def. $OPT(i, w)$ = max profit subset of items $1, \ldots, i$ with weight limit w.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using weight limit w.

optimal substructure property
(proof via exchange argument)

Case 2. OPT selects item i.

- New weight limit $= w - w_i$.
- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using this new weight limit.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\left\{ OPT(i-1, w),\quad v_i + OPT(i-1, w-w_i) \right\} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

  FOR $w = 0$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w-w_i] \}.$

RETURN $M[n, W]$.

Knapsack problem: bottom-up demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i) \} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

Knapsack problem: running time

Theorem. There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(n\ W)$ time and $\Theta(n\ W)$ space.

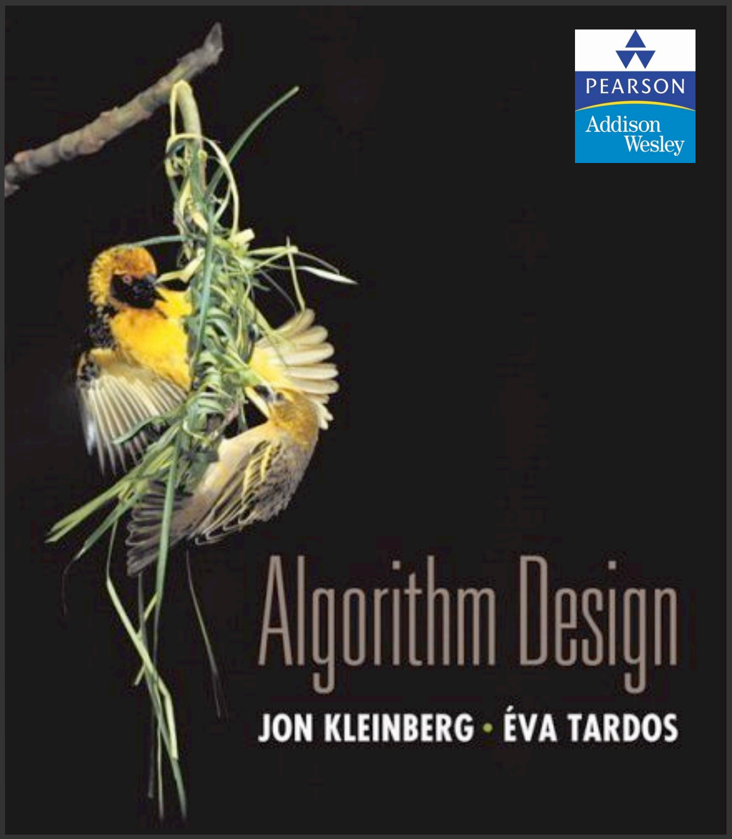
weights are integers between 1 and W

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\ W)$ table entries.
- After computing optimal values, can trace back to find solution: take item i in $OPT(i, w)$ iff $M[i, w] > M[i-1, w]$. ▪

Remarks.

- Not polynomial in input size! ← "pseudo-polynomial"
- Decision version of knapsack problem is NP-COMPLETE. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]



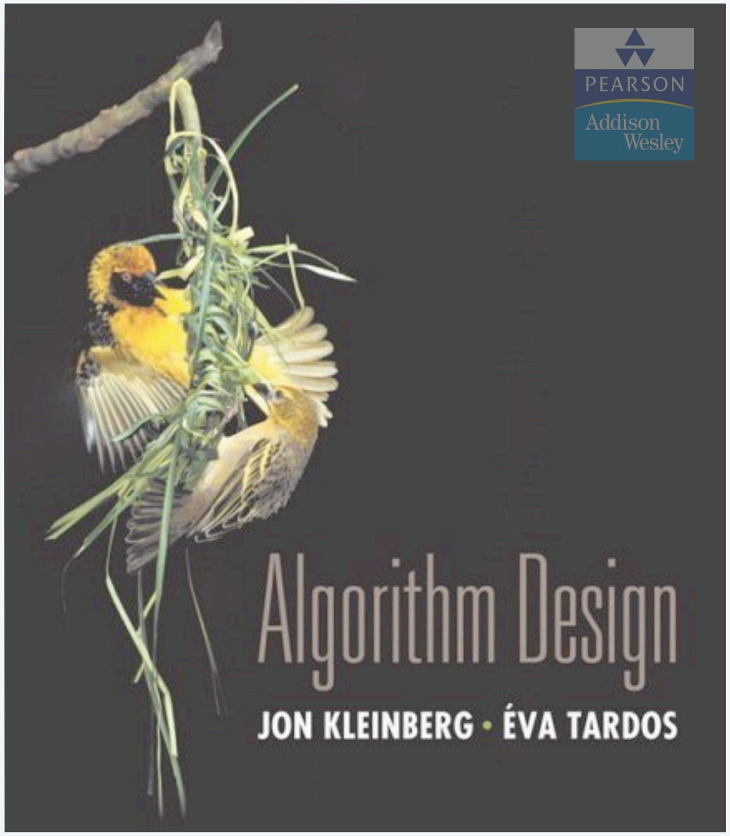
6. Dynamic Programming II

- *sequence alignment*
- *Hirschberg's algorithm*
- *Bellman-Ford algorithm*
- *distance vector protocols*
- *negative cycles in a digraph*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on Oct 11, 2014, 9:26 AM



SECTION 6.6

6. DYNAMIC PROGRAMMING II

- *sequence alignment*
- *Hirschberg's algorithm*
- *Bellman-Ford algorithm*
- *distance vector protocols*
- *negative cycles in a digraph*

String similarity

Q. How similar are two strings?

Ex. ocurrance and occurrence.

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

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.

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

cost = $\delta + \alpha_{CG} + \alpha_{TA}$

Applications. Unix diff, speech recognition, computational biology, ...

Sequence alignment

Goal. Given two strings $x_1\, x_2 \ldots x_m$ and $y_1\, y_2 \ldots y_n$ find min cost alignment.

Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

$x_i - y_j$ and $x_{i'} - y_{j'}$ cross if $i < i'$, but $j > j'$

Def. The cost of an alignment M is:

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

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	–	G
–	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

an alignment of CTACCG and TACATG:

$$M = \{\, x_2–y_1,\, x_3–y_2,\, x_4–y_3,\, x_5–y_4,\, x_6–y_6 \,\}$$

Sequence alignment: problem structure

Def. $OPT(i, j)$ = min cost of aligning prefix strings $x_1\, x_2\, \ldots\, x_i$ and $y_1\, y_2\, \ldots\, y_j$.

Case 1. OPT matches $x_i - y_j$.
Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1\, x_2\, \ldots\, x_{i-1}$ and $y_1\, y_2\, \ldots\, y_{j-1}$.

Case 2a. OPT leaves x_i unmatched.
Pay gap for x_i + min cost of aligning $x_1\, x_2\, \ldots\, x_{i-1}$ and $y_1\, y_2\, \ldots\, y_j$.

optimal substructure property
(proof via exchange argument)

Case 2b. OPT leaves y_j unmatched.
Pay gap for y_j + min cost of aligning $x_1\, x_2\, \ldots\, x_i$ and $y_1\, y_2\, \ldots\, y_{j-1}$.

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

Sequence alignment: algorithm

SEQUENCE-ALIGNMENT $(m, n, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha)$

FOR $i = 0$ TO m

$\quad M[i, 0] \leftarrow i\,\delta.$

FOR $j = 0$ TO n

$\quad M[0, j] \leftarrow j\,\delta.$

FOR $\mathrm{i} = 1$ TO m

$\quad$ FOR $\mathrm{j} = 1$ TO n

$$M[i, j] \leftarrow \min \{\, \alpha[x_i, y_j] + M[i-1, j-1],\ \delta + M[i-1, j],\ \delta + M[i, j-1]).$$

RETURN $M[m, n]$.

Sequence alignment: analysis

Theorem. The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length m and n in $\Theta(mn)$ time and $\Theta(mn)$ space.

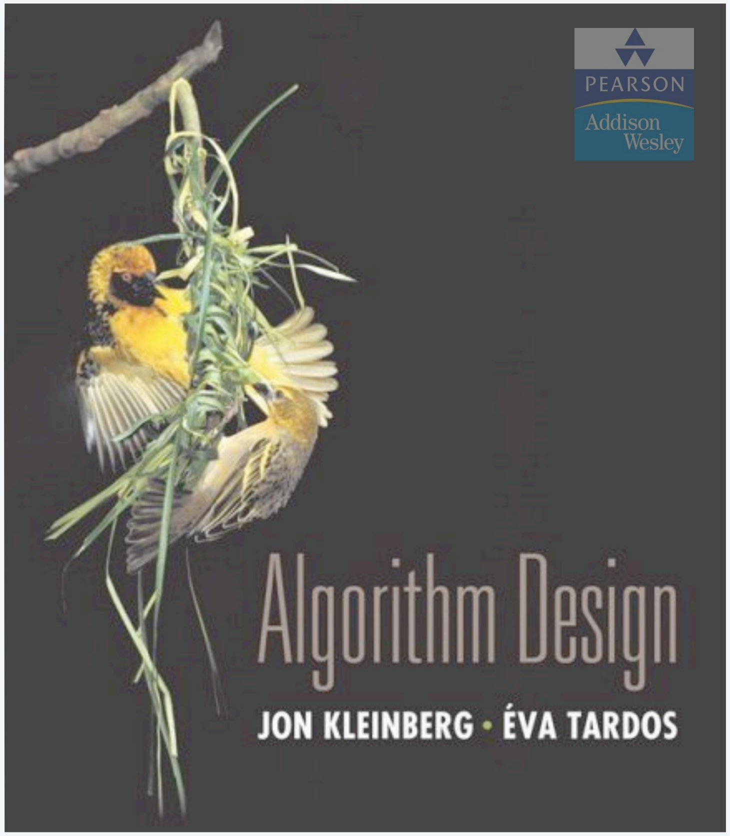
Pf.

- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ▪

Q. Can we avoid using quadratic space?

A. Easy to compute optimal value in $O(mn)$ time and $O(m + n)$ space.

- Compute $\text{OPT}(i, \bullet)$ from $\text{OPT}(i - 1, \bullet)$.
- But, no longer easy to recover optimal alignment itself.



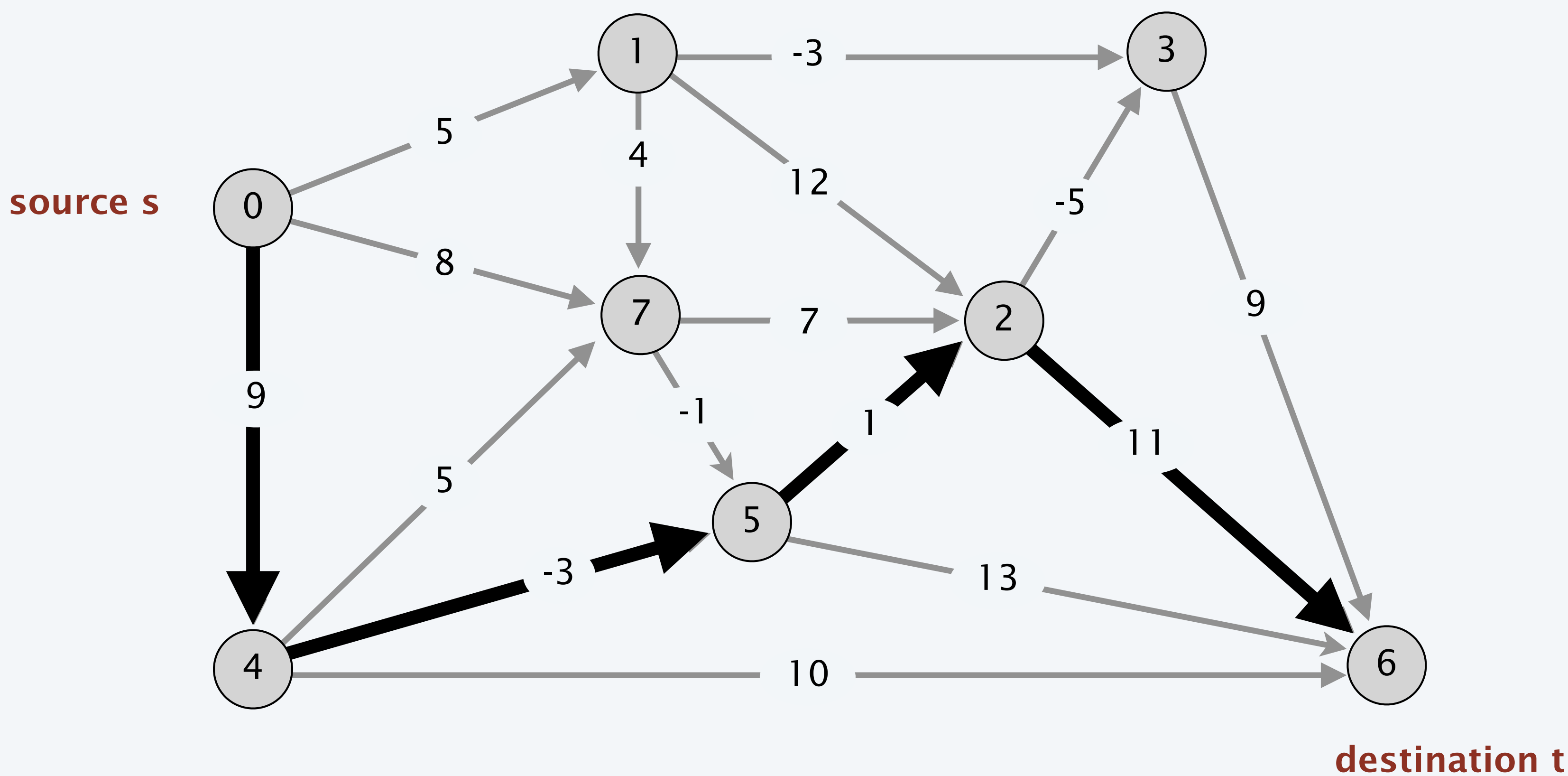
SECTION 6.8

6. DYNAMIC PROGRAMMING II

- *sequence alignment*
- *Hirschberg's algorithm*
- *Bellman-Ford*
- *distance vector protocols*
- *negative cycles in a digraph*

Shortest paths

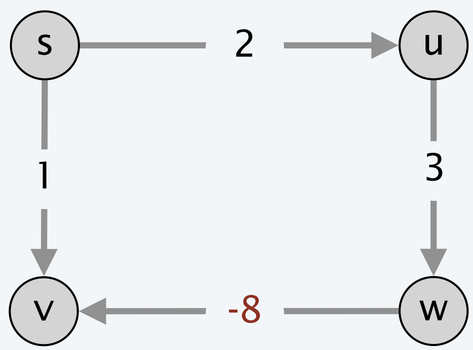
Shortest path problem. Given a digraph $G = (V, E)$, with arbitrary edge weights or costs c_{vw}, find cheapest path from node s to node t.



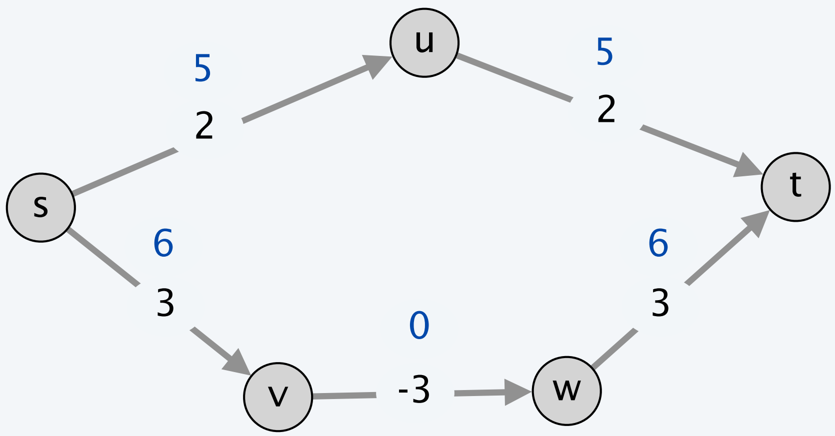
cost of path = 9 – 3 + 1 + 11 = 18

Shortest paths: failed attempts

Dijkstra. Can fail if negative edge weights.

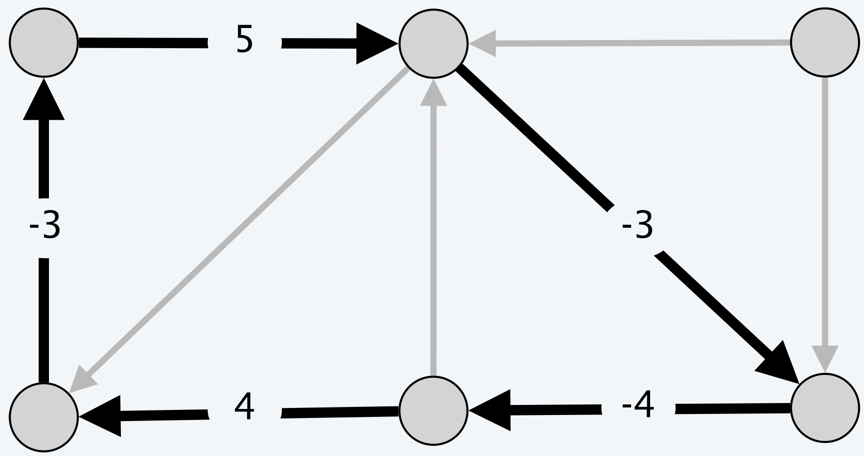


Reweighting. Adding a constant to every edge weight can fail.



Negative cycles

Def. A negative cycle is a directed cycle such that the sum of its edge weights is negative.

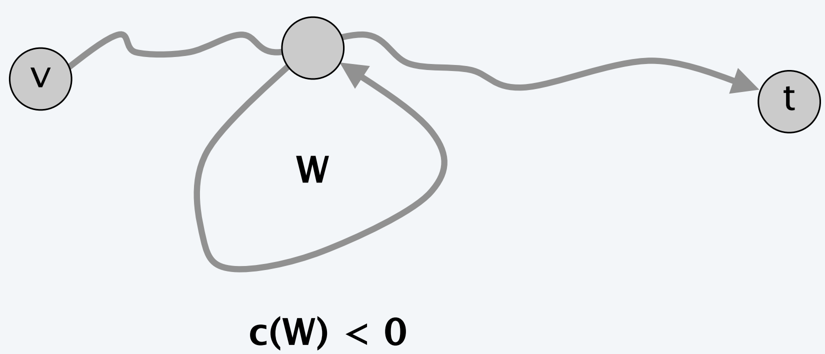


a negative cycle W : $c(W) \ = \ \sum_{e \in W} c_e \ < \ 0$

Shortest paths and negative cycles

Lemma 1. If some path from v to t contains a negative cycle, then there does not exist a cheapest path from v to t.

Pf. If there exists such a cycle W, then can build a $v \rightsquigarrow t$ path of arbitrarily negative weight by detouring around cycle as many times as desired. ▪

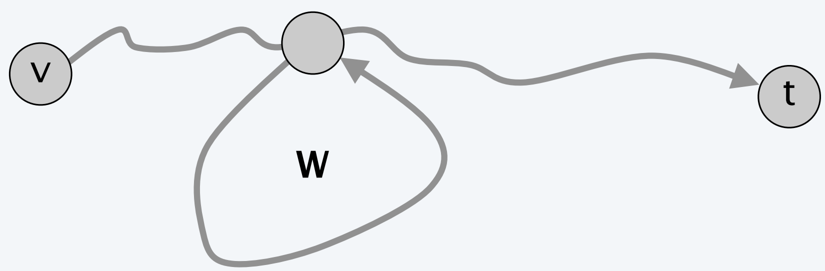


Shortest paths and negative cycles

Lemma 2. If G has no negative cycles, then there exists a cheapest path from v to t that is simple (and has $\leq n - 1$ edges).

Pf.

- Consider a cheapest $v \rightarrow t$ path P that uses the fewest number of edges.
- If P contains a cycle W, can remove portion of P corresponding to W without increasing the cost. ▪

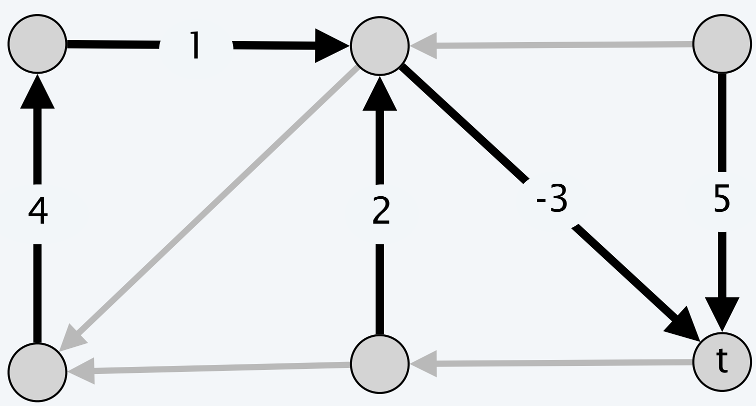


c(W) ≥ 0

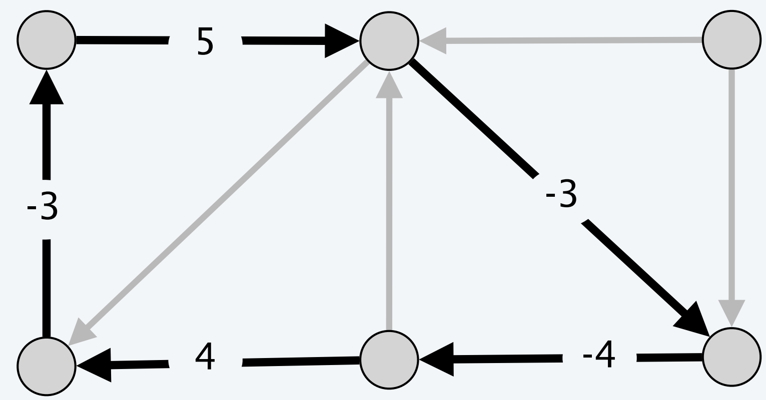
Shortest path and negative cycle problems

Shortest path problem. Given a digraph $G = (V, E)$ with edge weights c_{vw} and no negative cycles, find cheapest $v \rightarrow t$ path for each node v.

Negative cycle problem. Given a digraph $G = (V, E)$ with edge weights c_{vw}, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Shortest paths: dynamic programming

Def. $OPT(i, v)$ = cost of shortest $v \leadsto t$ path that uses $\leq i$ edges.

- Case 1: Cheapest $v \leadsto t$ path uses $\leq i - 1$ edges.
  - $OPT(i, v) = \text{OPT}(i - 1, v)$

optimal substructure property
(proof via exchange argument)

- Case 2: Cheapest $v \leadsto t$ path uses exactly i edges.
  - if (v, w) is first edge, then OPT uses (v, w), and then selects best $w \leadsto t$ path using $\leq i - 1$ edges

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

Observation. If no negative cycles, $OPT(n - 1, v)$ = cost of cheapest $v \leadsto t$ path.

Pf. By Lemma 2, cheapest $v \leadsto t$ path is simple. ▪

Shortest paths: implementation

SHORTEST-PATHS (V, E, c, t)

FOREACH node $v \in V$

  $M[0, v] \leftarrow \infty$.

$M[0, t] \leftarrow 0$.

FOR i = 1 TO $n - 1$

  FOREACH node $v \in V$

    $M[i, v] \leftarrow M[i-1, v]$.

    FOREACH edge $(v, w) \in E$

      $M[i, v] \leftarrow \min \{ M[i, v],\ M[i-1, w] + c_{vw} \}$.

Shortest paths: implementation

Theorem 1. Given a digraph $G = (V, E)$ with no negative cycles, the dynamic programming algorithm computes the cost of the cheapest $v \rightarrow t$ path for each node v in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Pf.

- Table requires $\Theta(n^2)$ space.
- Each iteration i takes $\Theta(m)$ time since we examine each edge once. ▪

Finding the shortest paths.

- Approach 1: Maintain a $successor(i, v)$ that points to next node on cheapest $v \rightarrow t$ path using at most i edges.
- Approach 2: Compute optimal costs $M[i, v]$ and consider only edges with $M[i, v] = M[i-1, w] + c_{vw}$.

Shortest paths: practical improvements

Space optimization. Maintain two 1d arrays (instead of 2d array).

- $d(v)$ = cost of cheapest $v{\rightarrow}t$ path that we have found so far.
- $successor(v)$ = next node on a $v{\rightarrow}t$ path.

Performance optimization. If $d(w)$ was not updated in iteration $i-1$, then no reason to consider edges entering w in iteration i.

Bellman-Ford: efficient implementation

BELLMAN-FORD (V, E, c, t)

FOREACH node $v \in V$

$d(v) \leftarrow \infty$.

$successor(v) \leftarrow null$.

$d(t) \leftarrow 0$.

FOR i = 1 TO $n - 1$

FOREACH node $w \in V$

IF ($d(w)$ was updated in previous iteration)

FOREACH edge $(v, w) \in E$

IF ($d(v) > d(w) + c_{vw}$)

$d(v) \leftarrow d(w) + c_{vw}$.

$successor(v) \leftarrow w$.

IF no $d(w)$ value changed in iteration i, STOP.

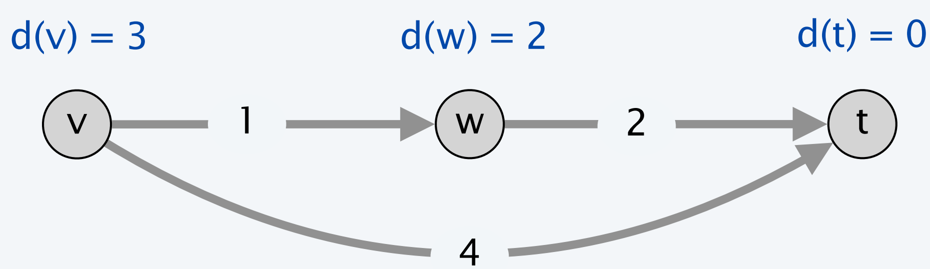
---

1 pass

Bellman-Ford: analysis

Claim. After ~~the i^{th} pass of Bellman-Ford, $d(v)$ equals the cost of the cheapest $v \rightarrow t$ path using at most i edges~~.

Counterexample. Claim is false!



**if nodes w considered before node v,
then d(v) = 3 after 1 pass**

Bellman-Ford: analysis

Lemma 3. Throughout Bellman-Ford algorithm, $d(v)$ is the cost of some $v \to t$ path; after the i^{th} pass, $d(v)$ is no larger than the cost of the cheapest $v \to t$ path using $\leq i$ edges.

Pf. [by induction on i]

- Assume true after i^{th} pass.
- Let P be any $v \to t$ path with $i + 1$ edges.
- Let (v, w) be first edge on path and let P' be subpath from w to t.
- By inductive hypothesis, $d(w) \leq c(P')$ since P' is a $w \to t$ path with i edges.
- After considering v in pass i+1:

$$\begin{aligned} d(v) &\leq c_{vw} + d(w) \\ &\leq c_{vw} + c(P') \\ &= c(P) \quad \blacksquare \end{aligned}$$

Theorem 2. Given a digraph with no negative cycles, Bellman-Ford computes the costs of the cheapest $v \to t$ paths in $O(mn)$ time and $\Theta(n)$ extra space.

Pf. Lemmas 2 + 3. ▪

can be substantially faster in practice

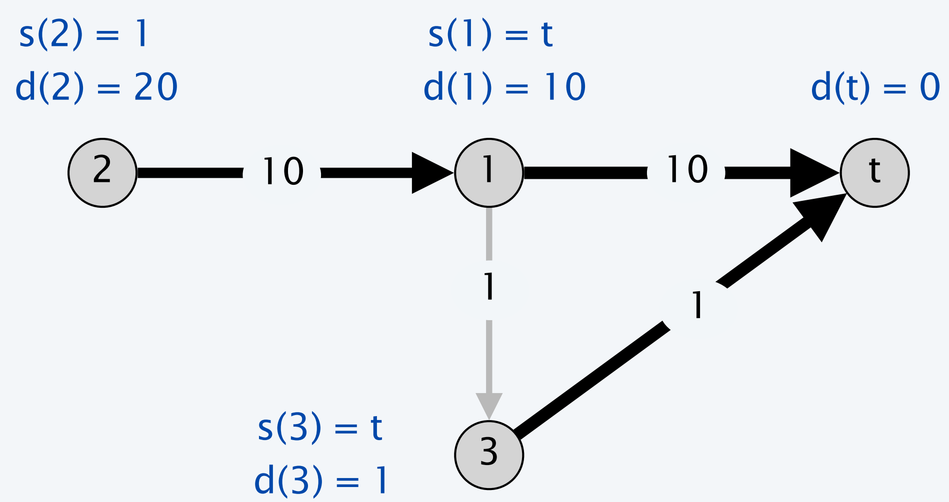
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following $successor(v)$ pointers gives a directed path from v to t of cost $d(v)$.~~

Counterexample. Claim is false!

- Cost of successor $v \rightsquigarrow t$ path may have strictly lower cost than $d(v)$.

consider nodes in order: t, 1, 2, 3



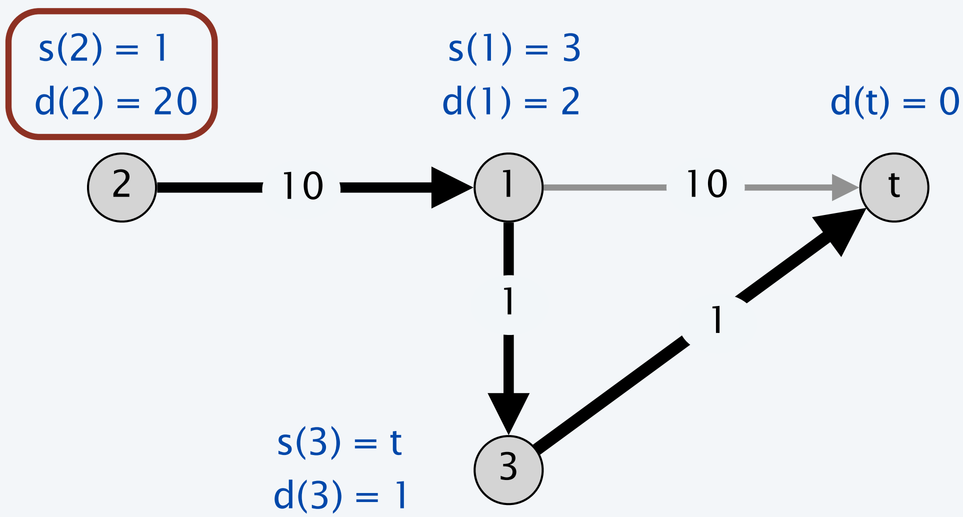
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following $successor(v)$ pointers gives a directed path from v to t of cost $d(v)$.~~

Counterexample. Claim is false!

- Cost of successor $v \rightsquigarrow t$ path may have strictly lower cost than $d(v)$.

consider nodes in order: t, 1, 2, 3



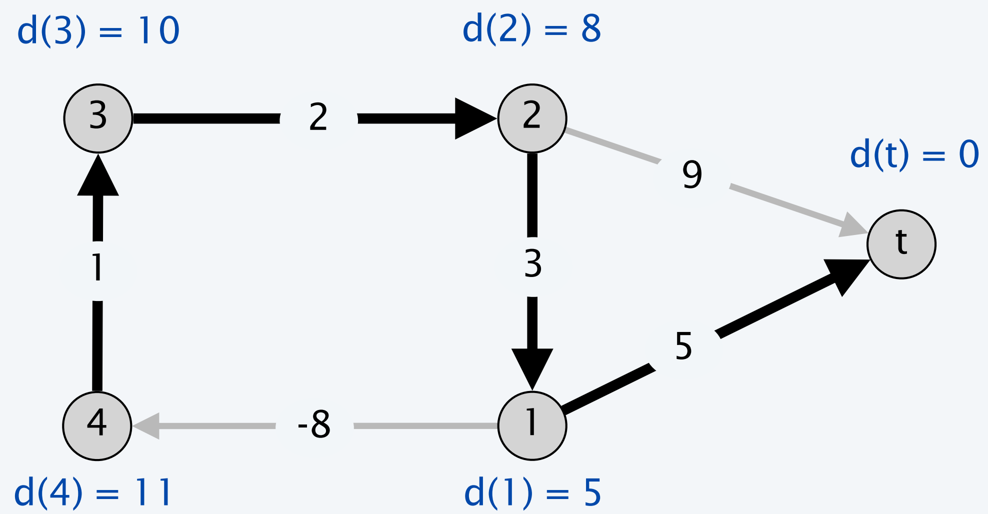
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(*v*) pointers gives a directed path from *v* to *t* of cost *d*(*v*).~~

Counterexample. Claim is false!

- Cost of successor $v \rightsquigarrow t$ path may have strictly lower cost than $d(v)$.
- Successor graph may have cycles.

consider nodes in order: t, 1, 2, 3, 4



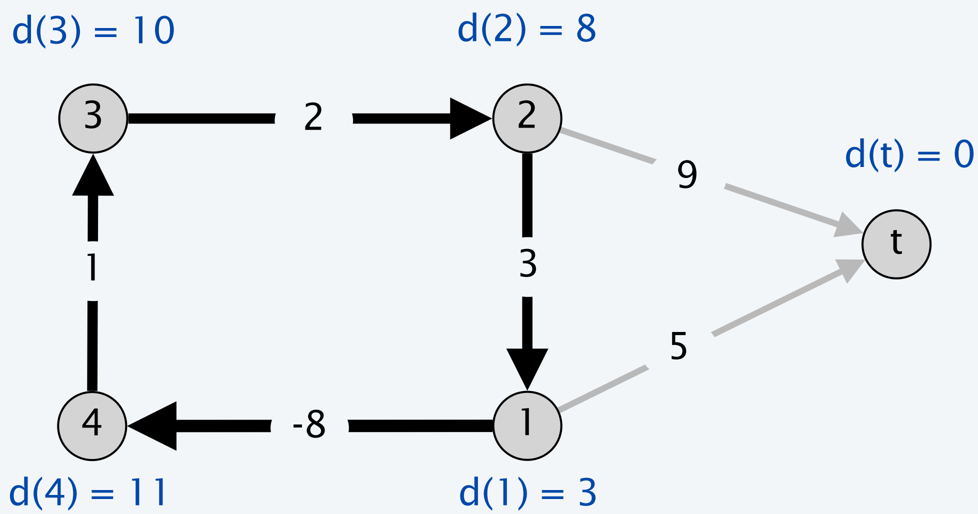
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(*v*) pointers gives a directed path from *v* to *t* of cost *d*(*v*).~~

Counterexample. Claim is false!

- Cost of successor $v \rightarrow t$ path may have strictly lower cost than $d(v)$.
- Successor graph may have cycles.

consider nodes in order: t, 1, 2, 3, 4



Bellman-Ford: finding the shortest path

Lemma 4. If the successor graph contains a directed cycle W, then W is a negative cycle.

Pf.

- If $successor(v) = w$, we must have $d(v) \geq d(w) + c_{vw}$.
  (LHS and RHS are equal when $successor(v)$ is set; $d(w)$ can only decrease; $d(v)$ decreases only when $successor(v)$ is reset)
- Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ be the nodes along the cycle W.
- Assume that (v_k, v_1) is the last edge added to the successor graph.
- Just prior to that:

$$
\begin{array}{lcll}
d(v_1) & \geq & d(v_2) & + \; c(v_1, v_2) \\
d(v_2) & \geq & d(v_3) & + \; c(v_2, v_3) \\
\vdots & \vdots & & \vdots \\
d(v_{k-1}) & \geq & d(v_k) & + \; c(v_{k-1}, v_k) \\
d(v_k) & > & d(v_1) & + \; c(v_k, v_1)
\end{array}
$$

holds with strict inequality since we are updating $d(v_k)$

- Adding inequalities yields $\underbrace{c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k) + c(v_k, v_1) < 0}_{\text{W is a negative cycle}}$. ▪

Bellman-Ford: finding the shortest path

Theorem 3. Given a digraph with no negative cycles, Bellman-Ford finds the cheapest $s \rightsquigarrow t$ paths in $O(mn)$ time and $\Theta(n)$ extra space.

Pf.

- The successor graph cannot have a negative cycle. [Lemma 4]
- Thus, following the successor pointers from s yields a directed path to t.
- Let $s = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = t$ be the nodes along this path P.
- Upon termination, if $successor(v) = w$, we must have $d(v) = d(w) + c_{vw}$.
  (LHS and RHS are equal when $successor(v)$ is set; $d(\cdot)$ did not change)

  since algorithm terminated
- Thus,

$$\begin{aligned} d(v_1) &= d(v_2) &&+ c(v_1, v_2) \\ d(v_2) &= d(v_3) &&+ c(v_2, v_3) \\ &\vdots && \vdots \\ d(v_{k-1}) &= d(v_k) &&+ c(v_{k-1}, v_k) \end{aligned}$$

- Adding equations yields $\underset{\text{min cost of any } s \rightsquigarrow t \text{ path (Theorem 2)}}{d(s)} = \underset{0}{d(t)} + \underbrace{c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k)}_{\text{cost of path P}}$. ▪

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2 m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2 m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

Improved algorithms:

- Floyd-Warshall: $O(n^3)$

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2 m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

Improved algorithms:

- Floyd-Warshall: $O(n^3)$
- Johnson: $O(n^2 \lg n + nm)$ (really cool!)
    - Run Bellman-Ford from an arbitrary vertex s in $O(nm)$ time.
    - Change edge weights so they are all non-negative but shortest paths don't change!
    - Run Dijkstra n times.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

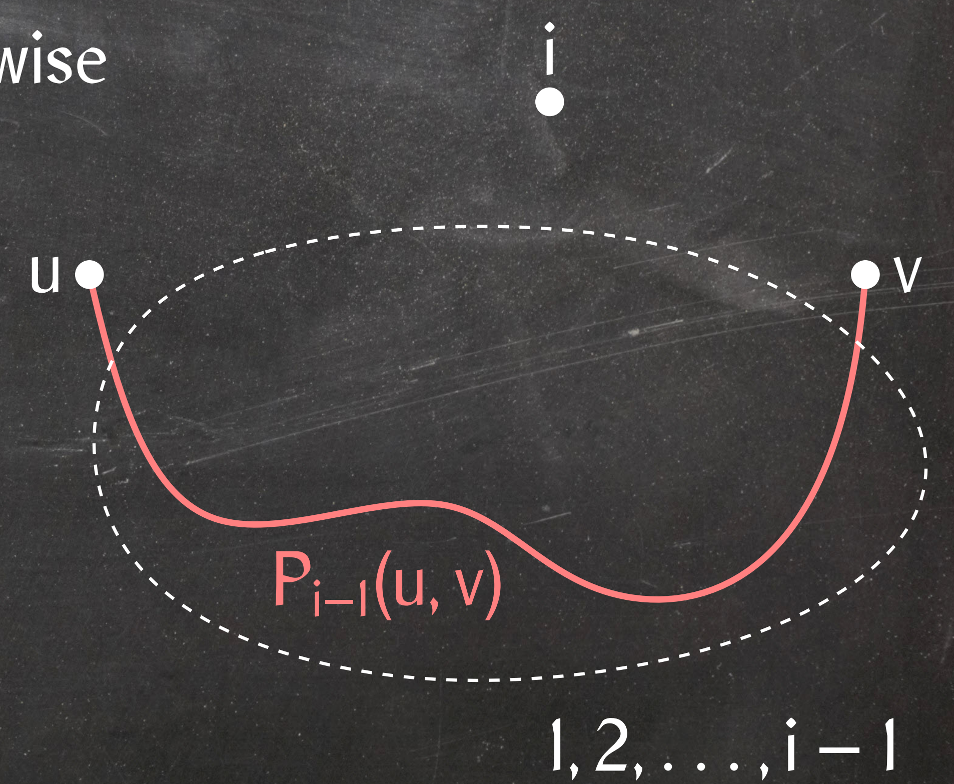
$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

If $i \notin P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, v)$.



All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

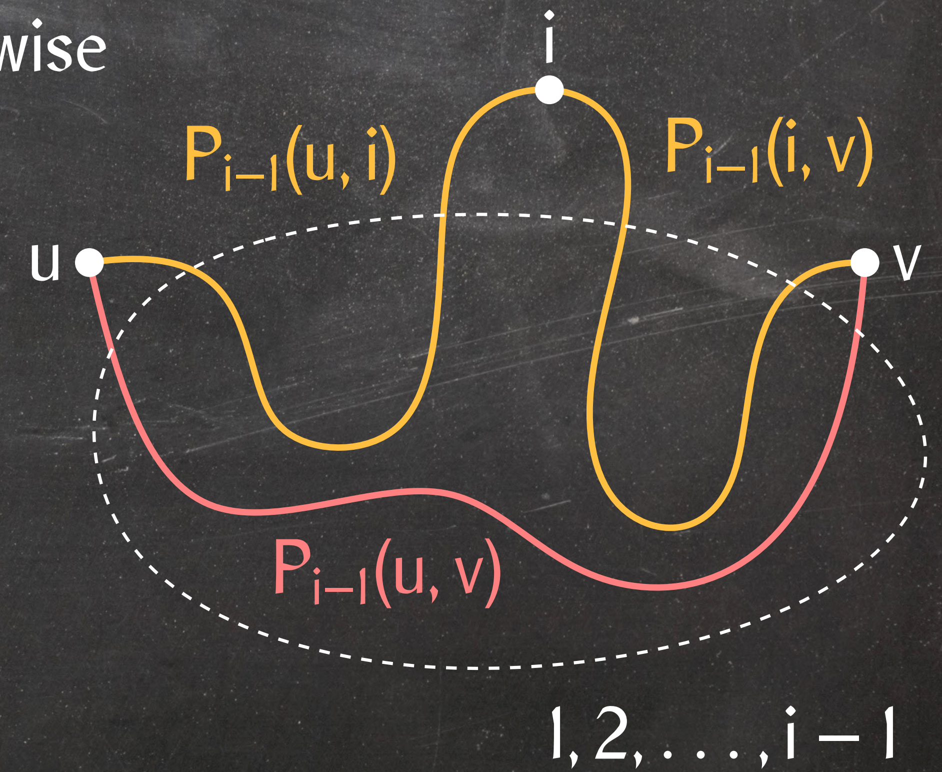
$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

- If $i \notin P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, v)$.
- If $i \in P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, i) \circ P_{i-1}(i, v)$.



All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u,v)$ be the length of the shortest path $P_i(u,v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u,v) = d_n(u,v)$

If $i = 0$, $P_0(u,v)$ cannot visit any vertices other than u and v:

$$d_0(u,v) = \begin{cases} w(u,v) & (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u,v)$ includes vertex i or it doesn't.

If $i \notin P_i(u,v)$, then $P_i(u,v) = P_{i-1}(u,v)$.

If $i \in P_i(u,v)$, then $P_i(u,v) = P_{i-1}(u,i) \circ P_{i-1}(i,v)$.

$$d_i(u,v) = \min(d_{i-1}(u,v),\, d_{i-1}(u,i) + d_{i-1}(i,v))$$

All-Pairs Shortest Paths: The Floyd-Warshall Algorithm

FloydWarshall(G)

```
for every pair of vertices u, v ∈ G
    do d[u, v] = ∞
       p[u, v] = Nothing
for every vertex v ∈ G
    do d[v, v] = 0
       p[v, v] = v
for every edge e ∈ G
    do d[e.tail, e.head] = e.weight
       p[e.tail, e.head] = e.tail
for i = 1 to n
    do for every pair of vertices u, v ∈ G such that i ∉ {u, v}
           do if d[u, v] > d[u, i] + d[i, v]
                 then d[u, v] = d[u, i] + d[i, v]
                      p[u, v] = p[i, v]
return (d, p)
```

All-Pairs Shortest Paths: The Floyd-Warshall Algorithm

FloydWarshall(G)

```
for every pair of vertices u, v ∈ G
    do d[u, v] = ∞
       p[u, v] = Nothing
for every vertex v ∈ G
    do d[v, v] = 0
       p[v, v] = v
for every edge e ∈ G
    do d[e.tail, e.head] = e.weight
       p[e.tail, e.head] = e.tail
for i = 1 to n
    do for every pair of vertices u, v ∈ G such that i ∉ {u, v}
           do if d[u, v] > d[u, i] + d[i, v]
                 then d[u, v] = d[u, i] + d[i, v]
                      p[u, v] = p[i, v]
return (d, p)
```

ReportPath(p, u, v)

```
if p[u, v] = Nothing
    then return Nothing
P = [v]
while v ≠ u
    do v = p[u, v]
       P.prepend(v)
return P
```