CS 341: Algorithms

University of Waterloo
Éric Schost
eschost@uwaterloo.ca

Module 5: dynamic programming

Goals

This module: the dynamic programming paradigm through examples

- interval scheduling, longest increasing subsequence, longest common subsequence, etc

Computational model:

- word RAM
- assume all weights, values, capacities, deadlines, etc, fit in a word

What about the name?

- **programming** as in **decision making**
- **dynamic** because it sounds cool.

Warmup example: Fibonacci numbers

A slow recursive algorithm

Def: Fibonacci numbers

- $F_0 = 0$, $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

```
Fib(n)
1.    if n = 0 return 0
2.    if n = 1 return 1
3.    return Fib(n − 1) + Fib(n − 2)
```

Assuming we count additions **at unit cost**, runtime is

$$T(0) = T(1) = 0, \quad T(n) = T(n-1) + T(n-2) + 1$$

This gives $T(n) = F(n+1) - 1$, so $\boldsymbol{T(n) \in \Theta(\varphi^n)}$, $\varphi = (1 + \sqrt{5})/2$.

A better algorithm

Observations

- to compute F_n, we only need the values of $F_0, \ldots, F_{n-1}$
- the algorithm recomputes them many, many times

A better algorithm

Observations

- to compute F_n, we only need the values of $F_0, \ldots, F_{n-1}$
- the algorithm recomputes them many, many times

Improved recursive algorithm

let $T = [0, 1, \bullet, \bullet, \ldots]$ be a global array
Fib(n)
1. **if** $T[n] = \bullet$
2. $\quad T[n] = \text{Fib}(n-1) + \text{Fib}(n-2)$
3. **return** $T[n]$

A better algorithm

Observations

- to compute F_n, we only need the values of $F_0, \ldots, F_{n-1}$
- the algorithm recomputes them many, many times

Iterative version

Fib(n)

```
1.    let T = [0, 1, •, •, ...]
2.    for i = 2, ..., n
3.        T[i] = T[i − 1] + T[i − 2]
4.    return T[n]
```

A better algorithm

Observations

- to compute F_n, we only need the values of $F_0, \ldots, F_{n-1}$
- the algorithm recomputes them many, many times

Iterative version (enhanced, not always feasible)

```
Fib(n)
1.    (u, v) ← (0, 1)
2.    for i = 2, ..., n
3.        (u, v) ← (v, u + v)
4.    return v
```

A better algorithm

Observations

- to compute F_n, we only need the values of $F_0, \ldots, F_{n-1}$
- the algorithm recomputes them many, many times

Iterative version (enhanced, not always feasible)

```
Fib(n)
1.    (u, v) ← (0, 1)
2.    for i = 2, ..., n
3.        (u, v) ← (v, u + v)
4.    return v
```

All these improved versions use $O(n)$ additions

Main feature: solve "subproblems" bottom up, and store solutions if needed.

Dynamic programming

Key features

- solve problems through recursion
- use a small (polynomial) number of **nested subproblems**
- may have to store results for all subproblems
- can often be turned into one (or more) loops

Dynamic programming vs divide-and-conquer

- dynamic programming usually deals with all input sizes $1, \ldots, n$
- DAC may not solve "subproblems"
- DAC algorithms not easy to rewrite iteratively

Interval scheduling

The problem

Input:

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- each interval has a weight $\boldsymbol{w_i}$

Output:

- a choice T of intervals that **do not overlap** and **maximizes** $\sum_{i \in T} w_i$
- greedy algorithm in the case $w_i = 1$

Example: A car rental company has the following requests for a given day:

- $I_1 = [2, 8]$, $w_1 = 6$
- $I_2 = [2, 4]$, $w_2 = 2$
- $I_3 = [5, 6]$, $w_3 = 1$
- $I_4 = [7, 9]$, $w_4 = 2$

Answer is $T = [I_1]$, $W = 6$

Sketch of the algorithm

Basic idea: either we choose I_n or not.

- then the optimum $\boldsymbol{O(I_1, \ldots, I_n)}$ is the max of two values:
- $\boldsymbol{w_n + O(I_{m_1}, \ldots, I_{m_s})}$, if we choose I_n, where $I_{m_1}, \ldots, I_{m_s}$ are the intervals that do not overlap with I_n
- $\boldsymbol{O(I_1, \ldots, I_{n-1})}$, if we don't choose I_n

In general, we don't know what $I_{m_1}, \ldots, I_{m_s}$ look like, so runtime may look like

$$T(n) = T(n-1) + T(n-2) + c \implies T(n) \text{ exponential.}$$

Goal:

- find a way to ensure that $I_{m_1}, \ldots, I_{m_s}$ are of the form $I_1, \ldots, I_s$, for some $s < n$ (and so on for all indices $< n$)
- then it suffices to optimize over all $I_1, \ldots, I_j$, $j = 1, \ldots, n$

Sorting intervals

Sorting $I_1, \ldots, I_n$ could help.

Attempt 1:

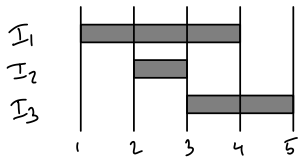
- we sort $I_1, \ldots, I_n$ by increasing **starting time**

Sorting intervals

Sorting $I_1, \ldots, I_n$ could help.

Attempt 1:

- we sort $I_1, \ldots, I_n$ by increasing **starting time**
- does not work

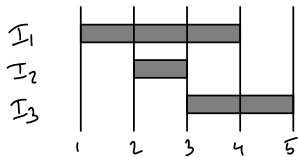


Sorting intervals

Sorting $I_1, \ldots, I_n$ could help.

Attempt 1:

- we sort $I_1, \ldots, I_n$ by increasing **starting time**
- does not work



Attempt 2:

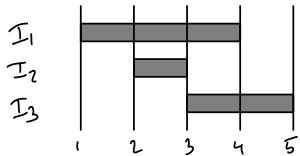
- we sort $I_1, \ldots, I_n$ by increasing **end time**

Sorting intervals

Sorting $I_1, \ldots, I_n$ could help.

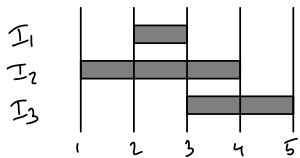
Attempt 1:

- we sort $I_1, \ldots, I_n$ by increasing **starting time**
- does not work



Attempt 2:

- we sort $I_1, \ldots, I_n$ by increasing **end time**
- works?



The indices p_j

Assume $I_1, \ldots, I_n$ sorted by increasing end time: $f_i \leq f_{i+1}$

Claim: for all j, the set of intervals $\boldsymbol{I_k \leq I_j}$ that do not overlap I_j is of the form $\boldsymbol{I_1, \ldots, I_{p_j}}$ for some $0 \leq p_j < j$ ($p_j = 0$ if no such interval)

The indices p_j

Assume $I_1, \ldots, I_n$ sorted by increasing end time: $f_i \leq f_{i+1}$

Claim: for all j, the set of intervals $\boldsymbol{I_k \leq I_j}$ that do not overlap I_j is of the form $\boldsymbol{I_1, \ldots, I_{p_j}}$ for some $0 \leq p_j < j$ ($p_j = 0$ if no such interval)

The algorithm will need the p_j's.

- if $-\infty \leq s_i < f_1$, $\boldsymbol{p_i = 0}$ f_1= earliest finish time
- if $f_1 \leq s_i < f_2$, $\boldsymbol{p_{A[i]} = 1}$
- ...

(we will write $f_0 = -\infty$)

Computing the p_j's

let A be a permutation of $[1, \ldots, n]$ such that

$$s_{A[1]} \leq s_{A[2]} \leq \cdots \leq s_{A[n]}$$

Exercise: make sure you know how to find such an A

FindPj$(A, s_1, \ldots, s_n, f_1, \ldots, f_n)$

```
1.    f_0 ← −∞
2.    i ← 1
3.    for k = 0, ..., n
4.        while i ≤ n and f_k ≤ s_{A[i]} < f_{k+1}
5.            p_i ← k
6.            i++
```

Runtime: $O(n \log(n))$ (sorting) and $O(n)$ (loops)

Main procedure

Definition: $\boldsymbol{M[i]}$ is the maximal weight we can get with intervals $I_1, \ldots, I_i$

Recurrence: $M[0] = 0$ and for $i \geq 1$

$$\boldsymbol{M[i] = \max(M[i-1], M[p_i] + w_i)}$$

Runtime: $\boldsymbol{O(n \log(n))}$ (sorting twice) and $\boldsymbol{O(n)}$ (finding the $M[i]$'s)

Exercise: recover the optimum set for an extra $O(n)$

Flipped version

Idea: we can discuss whether we take $\boldsymbol{I_1}$ or not. Call

$$N[i] = \text{ maximal weight we can get using } I_i, \ldots, I_n$$

Then $\boldsymbol{N[i]}$ is the maximum of

- $\boldsymbol{N[i+1]}$, if we **do not** choose I_i
- $\boldsymbol{N[\mathbf{next}_i] + w_i}$, where I_{next_i} is the first interval that starts after I_i ends

What we get: a similar induction, that finds the values of N in a decreasing manner. Same asymptotic runtime.

0/1 knapsack

The problem

Input:

- items $1, \ldots, n$ with **weights** $w_1, \ldots, w_n$ and **values** $v_1, \ldots, v_n$
- a **capacity** W

Output:

- a choice of items $S \subseteq \{1, \ldots, n\}$
- that satisfies the constraint $\sum_{i \in S} w_i \leq W$
- and maximizes the value $\sum_{i \in S} v_i$

The problem

Input:

- items $1, \ldots, n$ with **weights** $w_1, \ldots, w_n$ and **values** $v_1, \ldots, v_n$
- a **capacity** W

Output:

- a choice of items $S \subseteq \{1, \ldots, n\}$
- that satisfies the constraint $\sum_{i \in S} w_i \leq W$
- and maximizes the value $\sum_{i \in S} v_i$

Example:

- $w_1 = 3, w_2 = 4, w_3 = 6, w_4 = 5$
- $v_1 = 2, v_2 = 3, v_3 = 1, v_4 = 5$
- $W = 8$
- optimum $S = \{1, 4\}$ with weight 8 and value 7

The problem

Input:

- items $1, \ldots, n$ with **weights** $w_1, \ldots, w_n$ and **values** $v_1, \ldots, v_n$
- a **capacity** W

Output:

- a choice of items $S \subseteq \{1, \ldots, n\}$
- that satisfies the constraint $\sum_{i \in S} w_i \leq W$
- and maximizes the value $\sum_{i \in S} v_i$

See also:

- fractional knapsack (items can be divided), solved with a greedy algorithm

Setting up the recurrence

Basic idea: either we choose item n or not.

- then the optimum $\boldsymbol{O[n, W]}$ is the max of two values:
- $\boldsymbol{v_n + O[n-1, W-w_n]}$, if we choose n (and $w_n \leq W$)
- $\boldsymbol{O[n-1, W]}$, if we don't choose n

Initial conditions

- $O[0, w] = 0$ for any w

Algorithm

```
01KnapSack(v_1, ..., v_n, w_1, ..., w_n, W)
1.    initialize an array O[0..n, 0..W]
2.    O[0, w] = 0 for w = 0, ..., W
3.    for i = 1, ..., n
4.        for w = 0, ..., W
5.            if w_i ≤ w
6.                O[i, w] ← max(v_i + O[i − 1, w − w_i], O[i − 1, w])
7.            else
8.                O[i, w] ← O[i − 1, w]
```

Runtime $\Theta(nW)$.

Exercise: how to obtain the set S?

Discussion

This is called a **pseudo-polynomial** algorithm

- in our word RAM model, we have been assuming all v_is and w_is fit in a word
- so input size is $\Theta(n)$ words
- but the runtime also depends on the **values** of the inputs

01-knapsack is **NP-complete**, so we don't really expect to do much better

Another example: **count sort**, with inputs $A[1..n]$ and R, with all $0 \leq A[i] < R$

- create a size-R array B of linked lists
- for $i = 1, \ldots, n$, append i to $B[A[i]]$
- merge all $B[i]$'s into a final list

Runtime $\Theta(n + R)$

A related problem

Subset sum: given positive integers $a_1, \ldots, a_n$ and integer K, find $S \subseteq \{1, \ldots, n\}$ with

$$\sum_{i \in S} a_i = K$$

Option 1: write a "new" algorithm

- very much like knapsack
- pseudo polynomial runtime $O(nK)$

Option 2: use the knapsack algorithm with

- $w_1, \ldots, w_n = a_1, \ldots, a_n$
- $v_1, \ldots, v_n = a_1, \ldots, a_n$
- $W = K$

Longest increasing subsequence

The problem

Input: An array $A[1..n]$ of integers

Output: A **longest increasing subsequence** of A (or just its length)
(does **not** need to be contiguous)

Example: $A = [7, 1, 3, 10, 11, 5, 19]$ gives $[7, \mathbf{1}, \mathbf{3}, \mathbf{10}, \mathbf{11}, 5, \mathbf{19}]$

Remark: there are 2^n subsequences (including an empty one, which doesn't count)

Tentative subproblems

Attempt 1:

- **Subproblems:** the length $\ell[i]$ of a longest increasing subsequence of $A[1..i]$
- on the example, $\ell[6] = 4$
- so what? not enough to deduce $\ell[7]$

Attempt 2:

- **Subproblems:** the length $\ell[i]$ of a longest increasing subsequence of $A[1..i]$, together with its last entry
- example: $\ell[6] = 4$, with last element 11
- OK if we can add $A[i+1]$, but what if not?

A more complicated recurrence

Attempt 3:

- let $L[i]$ be the length of a longest increasing subsequence of $A[1..i]$ **that ends with** $A[i]$, for $i = 1, \ldots, n$
- so $L[1] = 1$

Idea:

- a longest increasing subsequence S ending at $A[i]$ looks like

$$S = [\ldots, A[j], A[i]] = S' \textbf{ cat } [A[i]]$$

- S' is a longest increasing subsequence ending at $A[j]$ (or it is empty)
- don't know j, but we can try all $j < i$ for which $A[j] < A[i]$

Iterative algorithm

LongestIncreasingSubsequence$(A[1..n])$

```
1.  L[1] ← 1
2.  for i = 2, ..., n do
3.      L[i] ← 1
4.      for j = 1, ..., i − 1 do
5.          if A[j] < A[i] then
6.              L[i] = max(L[i], L[j] + 1)
7.  return the maximum entry in L
```

Runtime: $\Theta(n^2)$

Remark:

- the algorithm does not return the sequence itself, but could be modified to do so

Bonus: a faster algorithm

As before, $\boldsymbol{\ell[i]}$=of a longest increasing subsequence of $A[1..i]$

Idea: we consider a list of "best" increasing sequences in $A[1..i]$

- one increasing sequence of length j for each $j = 1, \ldots, \ell[i]$
- for any j, a **best** increasing sequence of length j is one whose **last entry** is the **smallest**

Example: $A = [\mathbf{2}, \mathbf{8}, \mathbf{10}, \mathbf{11}, 1, 3, \mathbf{5}]$, $\boldsymbol{i = 6}$, $\boldsymbol{\ell[6] = 4}$

- $j = 1$, best increasing sequence $[1]$ — can add **5**
- $j = 2$, best increasing sequence $[1, 3]$ — can add **5**
- $j = 3$, best increasing sequence $[2, 8, 10]$ — can't add **5**
- $j = 4$, best increasing sequence $[2, 8, 10, 11]$ — can't add **5**

Bonus: a faster algorithm

As before, $\boldsymbol{\ell[i]}$=of a longest increasing subsequence of $A[1..i]$

Idea: we consider a list of "best" increasing sequences in $A[1..i]$

- one increasing sequence of length j for each $j = 1, \ldots, \ell[i]$
- for any j, a **best** increasing sequence of length j is one whose **last entry** is the **smallest**

Example: $A = [\mathbf{2}, \mathbf{8}, \mathbf{10}, \mathbf{11}, 1, 3, \mathbf{5}]$, $\boldsymbol{i = 6}$, $\boldsymbol{\ell[6] = 4}$

- $j = 1$, best increasing sequence $[\mathbf{1}]$ — can add **5**
- $j = 2$, best increasing sequence $[1, \mathbf{3}]$ — can add **5**
- $j = 3$, best increasing sequence $[2, 8, \mathbf{10}]$ — can't add **5**
- $j = 4$, best increasing sequence $[2, 8, 10, \mathbf{11}]$ — can't add **5**

$\mathbf{1 < 3 < 5 < 10 < 11}$ so $\boldsymbol{\ell[7] = 4}$ and we update the $j = 3$ sequence to $[1, 3, 5]$

Bonus: a faster algorithm

As before, $\boldsymbol{\ell[i]}$=of a longest increasing subsequence of $A[1..i]$

Idea: we consider a list of "best" increasing sequences in $A[1..i]$

- one increasing sequence of length j for each $j = 1, \ldots, \ell[i]$
- for any j, a **best** increasing sequence of length j is one whose **last entry** is the **smallest**

Example: $A = [\mathbf{2}, \mathbf{8}, \mathbf{10}, \mathbf{11}, 1, 3, \mathbf{15}]$, $\boldsymbol{i = 6}$, $\boldsymbol{\ell[6] = 4}$

- $j = 1$, best increasing sequence $[1]$ can add **15**
- $j = 2$, best increasing sequence $[1, 3]$ can add **15**
- $j = 3$, best increasing sequence $[2, 8, 10]$ can add **15**
- $j = 4$, best increasing sequence $[2, 8, 10, 11]$ can add **15**

Bonus: a faster algorithm

As before, $\boldsymbol{\ell[i]}$=of a longest increasing subsequence of $A[1..i]$

Idea: we consider a list of "best" increasing sequences in $A[1..i]$

- one increasing sequence of length j for each $j = 1, \ldots, \ell[i]$
- for any j, a **best** increasing sequence of length j is one whose **last entry** is the **smallest**

Example: $A = [\mathbf{2}, \mathbf{8}, \mathbf{10}, \mathbf{11}, 1, 3, \mathbf{15}]$, $\boldsymbol{i = 6}$, $\boldsymbol{\ell[6] = 4}$

- $j = 1$, best increasing sequence $[\mathbf{1}]$ can add **15**
- $j = 2$, best increasing sequence $[1, \mathbf{3}]$ can add **15**
- $j = 3$, best increasing sequence $[2, 8, \mathbf{10}]$ can add **15**
- $j = 4$, best increasing sequence $[2, 8, 10, \mathbf{11}]$ can add **15**

$\mathbf{1 < 3 < 10 < 11 < 15}$ so $\boldsymbol{\ell[7] = 5}$ and we have the $j = 5$ sequence $[2, 8, 10, 11, 15]$

Bonus: a faster algorithm

As before, $\boldsymbol{\ell[i]}$=of a longest increasing subsequence of $A[1..i]$

Idea: we consider a list of "best" increasing sequences in $A[1..i]$

- one increasing sequence of length j for each $j = 1, \ldots, \ell[i]$
- for any j, a **best** increasing sequence of length j is one whose **last entry** is the **smallest**

Example: $A = [\mathbf{2}, \mathbf{8}, \mathbf{10}, \mathbf{11}, 1, 3, \mathbf{0}]$, $\boldsymbol{i = 6}$, $\boldsymbol{\ell[6] = 4}$

- $j = 1$, best increasing sequence $[1]$ — can't add **0**
- $j = 2$, best increasing sequence $[1, 3]$ — can't add **0**
- $j = 3$, best increasing sequence $[2, 8, 10]$ — can't add **0**
- $j = 4$, best increasing sequence $[2, 8, 10, 11]$ — can't add **0**

Bonus: a faster algorithm

As before, $\boldsymbol{\ell[i]}$=of a longest increasing subsequence of $A[1..i]$

Idea: we consider a list of "best" increasing sequences in $A[1..i]$

- one increasing sequence of length j for each $j = 1, \ldots, \ell[i]$
- for any j, a **best** increasing sequence of length j is one whose **last entry** is the **smallest**

Example: $A = [\mathbf{2}, \mathbf{8}, \mathbf{10}, \mathbf{11}, 1, 3, \mathbf{0}]$, $\boldsymbol{i = 6}$, $\boldsymbol{\ell[6] = 4}$

- $j = 1$, best increasing sequence $[1]$ can't add **0**
- $j = 2$, best increasing sequence $[1, 3]$ can't add **0**
- $j = 3$, best increasing sequence $[2, 8, 10]$ can't add **0**
- $j = 4$, best increasing sequence $[2, 8, 10, 11]$ can't add **0**

$\mathbf{0 < 1 < 3 < 10 < 11}$ so $\boldsymbol{\ell[7] = 4}$ and we update the $j = 1$ sequence to $[0]$

Iterative algorithm

Remarks

- sufficient to store the last entry in each best increasing sequence
- these last entries are increasing ($\mathbf{1} < \mathbf{3} < \mathbf{10} < \mathbf{11}$)
- so we can use binary search to find where the new $A[i]$ fits

LongestIncreasingSubsequence$(A[1..n])$

1. $b \leftarrow [-\infty, \infty, \ldots, \infty]$, $\ell \leftarrow 0$ indexed starting from 0
2. **for** $i = 1, \ldots, n$ **do**
3. find $k \in \{0, \ldots, \ell\}$ such that $b[k] \leq A[i] \leq b[k+1]$
4. $b[k+1] \leftarrow A[i]$
5. **if** $k = \ell$, ℓ++
6. **return** ℓ

Runtime: $O(n \log(n))$

Longest common subsequence

The problem

Input: Arrays $A[1..n]$ and $B[1..m]$ of characters or integers

Output: The maximum length k of a common subsequence to A and B (subsequences do **not** need to be contiguous)

Example: $A =$**blurry**, $B =$**burger**, longest common subsequence is **burr**

Remark: there are 2^n subsequences in A, 2^m subsequences in B

Remark: this also solves the longest increasing subsequence problem

A bivariate recurrence

Definition: let $M[i, j]$ be the length of a longest subsequence between $\boldsymbol{A[1..i]}$ and $\boldsymbol{B[1..j]}$

- $M[0, j] = 0$ for all j
- $M[i, 0] = 0$ for all i
- $M[i, j]$ is the max of **up to three** values
  - $M[i, j-1]$ (don't use $B[j]$)
  - $M[i-1, j]$ (don't use $A[i]$)
  - **if** $A[i] = B[j]$, $\boldsymbol{1 + M[i-1, j-1]}$

The algorithm computes all $M[i, j]$, using two nested loops, so runtime $\boldsymbol{\Theta(mn)}$

Bonus: if $A[i] = B[j]$, no need to consider $M[i, j-1]$ and $M[i-1, j]$

Edit distance

The problem

Input: arrays $A[1..n]$ and $B[1..m]$ of characters

Output: minimum number of {add, delete, change} operations that turn A into B

Example: $A =$**snowy**, $B =$**sunny**

s	**n**	**o**	**w**	**y**		**s**	_	**n**	**o**	**w**	**y**	_	**s**	**n**	**o**	**w**	_	**y**
s	**u**	**n**	**n**	**y**		**s**	**u**	**n**	**n**	_	**y**	**s**	**u**	**n**	_	_	**n**	**y**
		3C					1A, 1C, 1D						2A, 1C, 2D					

Examples: DNA sequences made of **a**, **c**, **g**, **t**

The recurrence

Definition: let $\boldsymbol{D[i,j]}$ be the edit distance between $\boldsymbol{A[1..i]}$ and $\boldsymbol{B[1..j]}$

- $D[0,j] = j$ for all j (add j characters to empty A)
- $D[i,0] = i$ for all i (delete i characters from A)
- $D[i,j]$ is the min of **three** values
  - $D[i-1,j-1]$ (if $A[i] = B[j]$) or $D[i-1,j-1]+1$ (otherwise)
  - $D[i-1,j]+1$ (delete $A[i]$ and match $A[1..i-1]$ with $B[1..j]$)
  - $D[i,j-1]+1$ (add $B[j]$ and match $A[1..i]$ with $B[1..j-1]$)

The algorithm computes all $D[i,j]$, using two nested loops, so runtime $\boldsymbol{\Theta(mn)}$

Maximum independent set in a tree

The problem

Input:

- a tree $T = (V, E)$

Output:

- a maximum (sizewise) **independent set** in T
- a subset S of vertices is **independent** if there is **no edge** in T connecting two elements of S

Remark:

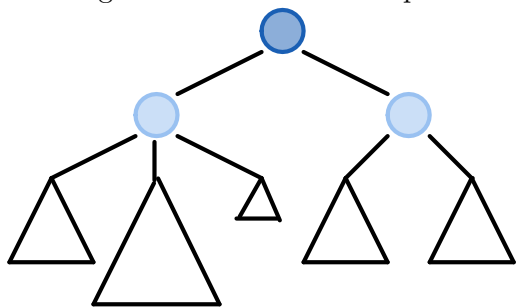
- in a general graph, IndependentSet is NP-complete

Setting up the recurrence

Case discussion: is the root r of T in S or not?

If yes:

- none of its children can be in S
- so we can look (recursively) at its **grandchildren**
- an independent set in a grandchild remains independent in T

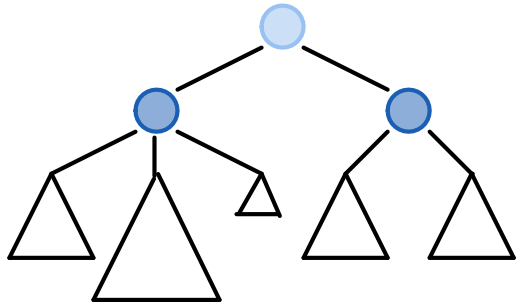


Setting up the recurrence

Case discussion: is the root r of T in S or not?

If no:

- all its children can be in S
- an independent set in a child remains independent in T
- so we look (recursively) at the children of r



Setting up the recurrence

Case discussion: is the root r of T in S or not?

Finally

$$O(T) = \max(1 + \sum_{C \text{ grandchild of } r} O(C), \sum_{C' \text{ child of } r} O(C'))$$

Runtime: proportional to

$$\sum_{v \in V} \#\text{children}(v) + \sum_{v \in V} \#\text{grandchildren}(v)$$

- first sum is number of vertices of level at least 1
- second sum is number of vertices of level at least 2
- so $\boldsymbol{O(n)}$ altogether

Exercise: find the independent set itself

Optimal binary search trees

The problem

Input:

- integers (or something else that can be ordered) $1, \ldots, n$
- probabilities of access $p_1, \ldots, p_n$, with $p_1 + \cdots + p_n = 1$

Output:

- an **optimal** BST with keys $1, \ldots, n$
- **optimal:** minimizes $\sum_{i=1}^{n} p_i \text{depth}(i)$ = expected number of tests for a search (here, depths start at 1)

The problem

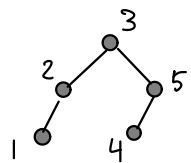
Input:

- integers (or something else that can be ordered) $1, \ldots, n$
- probabilities of access $p_1, \ldots, p_n$, with $p_1 + \cdots + p_n = 1$

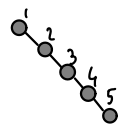
Output:

- an **optimal** BST with keys $1, \ldots, n$
- **optimal:** minimizes $\sum_{i=1}^{n} p_i \text{depth}(i)$ = expected number of tests for a search (here, depths start at 1)

Example: $p_1 = p_2 = p_3 = p_4 = p_5 = 1/5$



$1 \cdot \frac{1}{5} + 2 \cdot 2 \cdot \frac{1}{5} + 2 \cdot 3 \cdot \frac{1}{5} = \frac{11}{5}$



$\frac{1}{5}(1+2+3+4+5) = 3$

The problem

Input:

- integers (or something else that can be ordered) $1, \ldots, n$
- probabilities of access $p_1, \ldots, p_n$, with $p_1 + \cdots + p_n = 1$

Output:

- an **optimal** BST with keys $1, \ldots, n$
- **optimal:** minimizes $\sum_{i=1}^{n} p_i \text{depth}(i)$ = expected number of tests for a search (here, depths start at 1)

See also

- optimal static ordering for **linked lists**
- **Huffman trees**

both built using greedy algorithms

Setting up the recurrence

Definition define $\boldsymbol{M[i,j]}$ by

- $M[i,j]$ = the minimal cost for items $\{i,\ldots,j\}$, $1 \le i \le j \le n$
- $M[i,j] = 0$ for $j < i$

Recurrence

$$M[i,j] = \min_{i \le k \le j} \Big(\boldsymbol{M[i,k-1]} + \sum_{\boldsymbol{\ell=i}}^{\boldsymbol{k-1}} \boldsymbol{p_\ell} + \boldsymbol{p_k} + \boldsymbol{M[k+1,j]} + \sum_{\boldsymbol{\ell=k+1}}^{\boldsymbol{j}} \boldsymbol{p_\ell} \Big)$$

$$= \min_{i \le k \le j} \Big(M[i,k-1] + M[k+1,j] \Big) + \sum_{\ell=i}^{j} p_\ell$$

check: gives $M[i,i] = p_i$

Algorithm

Remark: to get $\sum_{\ell=i}^{j} p_\ell$:

- compute $S[\ell] = p_1 + \cdots + p_\ell$, for $\ell = 1, \ldots, n$
- then $p_i + \cdots + p_j = S[j] - S[i-1]$, with $S[0] = 0$

OptimalBST$(p_1, \ldots, p_n, S_0, \ldots, S_n)$

1. **for** $i = 1, \ldots, n+1$
2. $\quad M[i, i-1] \leftarrow 0$
3. **for** $d = 0, \ldots, n-1$ $\qquad d = j - i$
4. $\quad$ **for** $i = 1, \ldots, n-d$
5. $\quad\quad j \leftarrow d + i$
6. $\quad\quad M[i,j] \leftarrow \min_{i \le k \le j}(M[i, k-1] + M[k+1, j]) + S[j] - S[i-1]$

Runtime $O(n^3)$

A faster algorithm

For all i, j, let $\boldsymbol{k_{i,j}}$ be the (or any) index that gives the min at Step 6.

Claim (a bit difficult)

For all i, j, with $j > i$, we have $\boldsymbol{k_{i,j-1} \leq k_{i,j} \leq k_{i+1,j}}$

OptimalBST$(p_1, \ldots, p_n, S_0, \ldots, S_n)$

1. **for** $i = 1, \ldots, n+1$
2. $\quad M[i, i-1] \leftarrow 0$
3. **for** $d = 0, \ldots, n-1$ $\qquad d = j - i$
4. $\quad$ **for** $i = 1, \ldots, n-d$
5. $\quad\quad j \leftarrow d + i$
6. $\quad\quad$ **if** $d = 0$ **then** range $\leftarrow \{i\}$ **else** range $\leftarrow \boldsymbol{\{k_{i,j-1}, \ldots, k_{i+1,j}\}}$
7. $\quad\quad M[i,j], \boldsymbol{k_{i,j}} \leftarrow \min_{\boldsymbol{k} \in \text{range}}(M[i, k-1] + M[k+1, j]) + S[j] - S[i-1]$

Runtime, revisited

Work is proportional to

$$\begin{aligned}
\sum_{d=0}^{n-1}\sum_{i=1}^{n-d}(k_{i+1,j}-k_{i,j-1}+1) &= \sum_{d=0}^{n-1}\sum_{i=1}^{n-d}(k_{i+1,i+d}-k_{i,i+d-1}+1)\\
&\le \sum_{d=0}^{n-1}\sum_{i=1}^{n-d}(k_{i+1,i+d}-k_{i,i+d-1})+\sum_{d=0}^{n-1}\sum_{i=1}^{n-d}1\\
&\le \sum_{d=0}^{n-1}(k_{n-d+1,n}-k_{0,d-1})+\sum_{d=0}^{n-1}\sum_{i=1}^{n-d}1\\
&\le 2n^2
\end{aligned}$$

Conclusion: $\boldsymbol{\Theta(n^2)}$

Text segmentation (if time permits)

The problem

Input: a string, represented as an array $A[1..n]$

Output:

- **true** if we can **segment** of A into words
- **false** otherwise

(we assume that we can test if $A[i..j]$ is a word in $O(1)$ using **is_word**$[i..j]$)

Example: A=caramelow $\rightarrow$ **true**, with **car a me low**

Remark: there are 2^{n-1} ways to segment A

Subproblems and their recurrence

Subproblems: can we split $A[1..i]$ into words?

Definition: for $i = 1, \ldots, n$, let $s[i]$ be

- **true** if we can **segment** of $A[1..i]$ into words
- **false** otherwise

we set $s[0] =$ **true**

Recurrence: $s[i] = \textbf{or}_{j=0}^{i-1}\Big(s[j]\ \textbf{and}\ \textbf{is\_word}(A[j+1..i])\Big)$

Algorithm could be written recursively, but we'll focus on iterative version

A polynomial algorithm

```
IsSplittable(A[1..n])
1.      s[0] ← true
2.      for i = 1, ..., n do
3.          s[i] ← false
4.          for j = 0, ..., i − 1 do
5.              s[i] ← s[i] or (s[j] and is_word(A[j + 1..i]))
6.      return s[n]
```

Runtime: $\Theta(n^2)$

Remar:k the algorithm does not return the subdivision itself, but could be modified to do so

The Bellman-Ford algorithm

Outlook

Bellman-Ford

- given a **directed** graph $G = (V, E)$ with **weights** $w(e)$ on the edges
- assuming **no negative cycles**, want the shortest (=minimal weight) path / walk between a **source** s and **all vertices**

  (write $\delta(s, v)$ for the length of a minimal path = **distance** from s to v)
- can **detect** the existence of negative cycles
- very simple pseudo-code, but slower than Dijkstra's algorithm

Dynamic programming for shortest paths

Assume there is **no negative cycle** and **all vertices can be reached from** s

Definition:

- for $i = 0, \ldots, n$, set

$$\boldsymbol{\delta_i(s, v)} = \text{length of the shortest walk } \boldsymbol{s \rightsquigarrow v} \text{ with at most } \boldsymbol{i} \text{ edges}$$

this walk must be a path; if no walk, $\delta_i(s, v) = \infty$

Easy observations:

- $\delta_0(s, s) = 0$ and $\delta_0(s, v) = \infty$ for $v \neq s$
- $\delta_{n-1}(s, v) = \delta(s, v)$

Recurrence: $\boldsymbol{\delta_i(s, v) = \min(\delta_{i-1}(s, v), \min_{(u,v) \in E} \delta_{i-1}(s, u) + w(u, v))}$

Pseudo-code

BellmanFord(G, s)

1. $d_0 \leftarrow [0, \infty, \ldots, \infty]$ (s is the first index)
2. parent $\leftarrow [s, \bullet, \ldots, \bullet]$ (s is the first index)
3. **for** $i = 1, \ldots, n-1$ **do**
4. **for all** v in V **do**
5. $d_i[v] \leftarrow d_{i-1}[v]$
6. **for all** (u, v) in E **do**
7. **if** $d_{i-1}[u] + w(u, v) < d_i[v]$ **then**
8. $d_i[v] \leftarrow d_{i-1}[u] + w(u, v)$
9. parent$[v] \leftarrow u$

Correctness: $d_i[v] = \delta_i(s, v)$, so $d_{n-1}[v] = \delta(s, v)$

Runtime: $\boldsymbol{\Theta(mn)}$

Remark: need to loop over edges directed **toward** v

Saving space

Idea: use a single array d

BellmanFord2.0(G, s)

```
1.      d ← [0, ∞, ..., ∞] (s is the first index)
2.      parent ← [s, •, ..., •] (s is the first index)
3.      for i = 1, ..., n − 1 do
4.          for all (u, v) in E do
5.              if d[u] + w(u, v) < d[v] then
6.                  d[v] ← d[u] + w(u, v)
7.                  parent[v] ← u
```

Runtime: $\boldsymbol{\Theta(mn)}$

Correctness, part 1

Claim

For all i, after iteration i, we have $\boldsymbol{d[v] \leq d_i[v]}$ for all v

Idea: all quantities can only go down in version 2.0

Proof: by induction

- true for $i = 0$, so we suppose true at index $i - 1$ and prove true at i
- at the beginning of the loop, for all v, $\boldsymbol{d[v] \leq d_{i-1}[v]}$
- $d[v]$ can only decrease, so this stays true throughout the loop
- $d[v]$ is replaced by $\min(d[v], \min_{(u,v)\in E}(\square + w(u, v)))$, where $\square \leq d_{i-1}[u]$
- so at the end of iteration i, $\boldsymbol{d[v] \leq d_i[v]}$

Relaxations

The operation $\boldsymbol{d[v] \leftarrow \min(d[v], d[u] + w(u,v))}$ is called a **relaxation**

Claim

if $\boldsymbol{\delta(s,u) \leq d[u]}$ and $\boldsymbol{\delta(s,v) \leq d[v]}$ before relaxation, then $\boldsymbol{\delta(s,v) \leq d[v]}$ post-relaxation.

Proof

- $\delta(s,v) \leq \delta(s,u) + w(u,v)$ **(triangle inequality)**, so $\delta(s,v) \leq d[u] + w(u,v)$
- but also $\delta(s,v) \leq d[v]$

Consequence: if **all** $d[v]$ satisfy $\delta(s,v) \leq d[v]$, and we apply **any number** of relaxations, all inequalities stay true

Correctness, part 2

Claim

For $i = 0, \ldots, n-1$, after iteration i, $\boldsymbol{\delta(s,v) \leq d[v] \leq \delta_i(s,v)}$ for all v.

Proof:

- correctness part 1 gives $\boldsymbol{d[v] \leq \delta_i(s,v)}$
- previous slide gives $\boldsymbol{\delta(s,v) \leq d[v]}$

Summary

If there is no negative cycle

- at the end with $i = n - 1$, $\boldsymbol{d[v] = \delta(s, v)}$ for all v
- in particular, for any edge (u, v), $\boldsymbol{d[v] \leq d[u] + w(u, v)}$ (triangle inequality)

If there is a negative cycle $v_1, v_2, \ldots, v_k$, with $\boldsymbol{v_k = v_1}$

- **claim:** there must be an edge (v_i, v_{i+1}) with $d[v_{i+1}] > d[v_i] + w(v_i, v_{i+1})$
- **else:** $d[v_{i+1}] \leq d[v_i] + w(v_i, v_{i+1})$ for all i; sum and derive a contradiction

Conclusion: for extra $\boldsymbol{\Theta(m)}$, can check the presence of a negative cycle

Finding a negative cycle

Using the **first algorithm**:

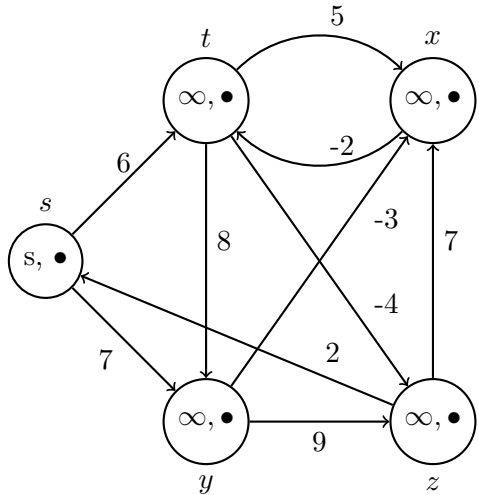
- if no negative cycle, $d_{n-1}(v) = d_n(v)$ for all v (easy)
- if $d_{n-1}(v) = d_n(v)$ for all v, no negative cycle (a bit more work)

If there is a negative cycle, there is v such that $d_n(v) < d_{n-1}(v)$.

Exercise

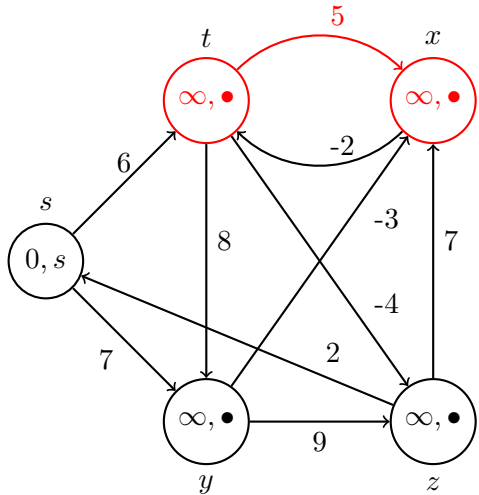
Then, there is a negative cycle on the path $s \to v$ stored in the `parent` array

Example: Bellman-Ford



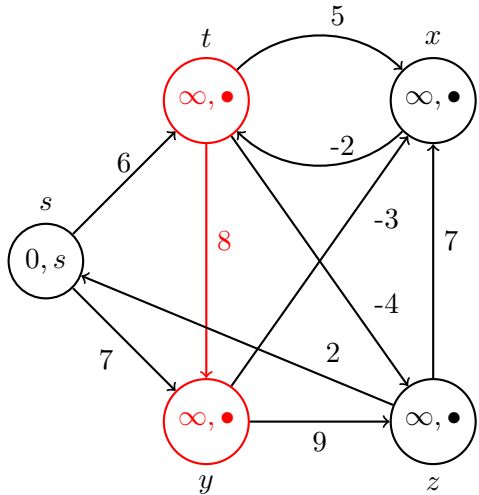
$i=1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Example: Bellman-Ford



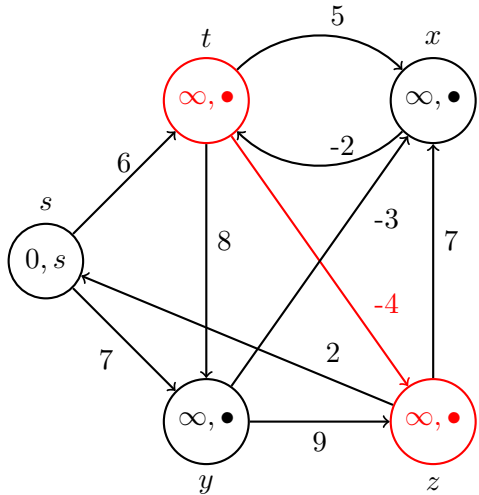
$i = 1$	(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y, z)	(z, x)	(z, s)	(s, t)	(s, y)
	×									

Example: Bellman-Ford



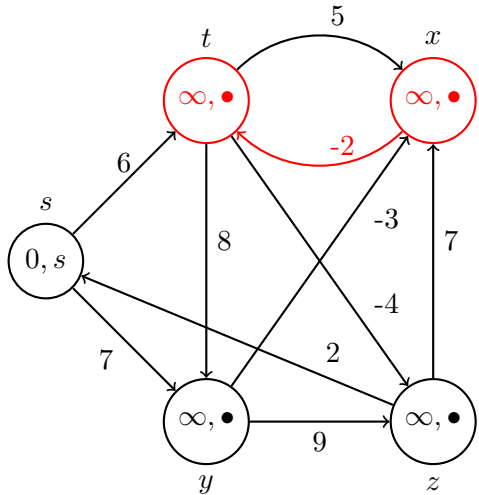
$i = 1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×								

Example: Bellman-Ford



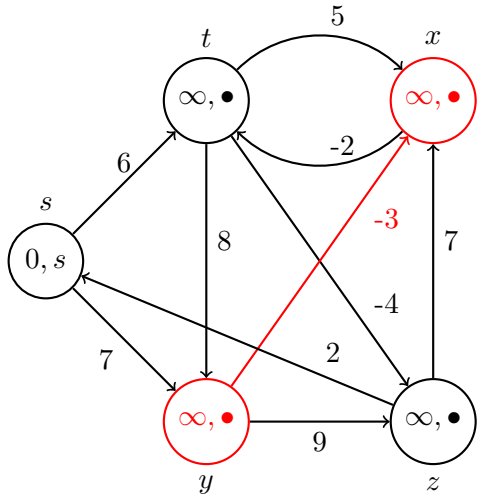
$i = 1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×							

Example: Bellman-Ford



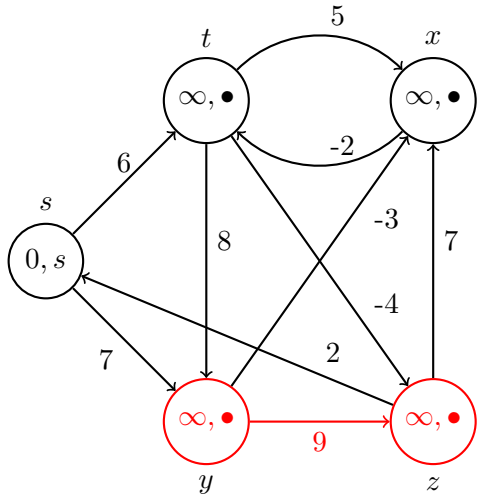
$i=1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	×						

Example: Bellman-Ford



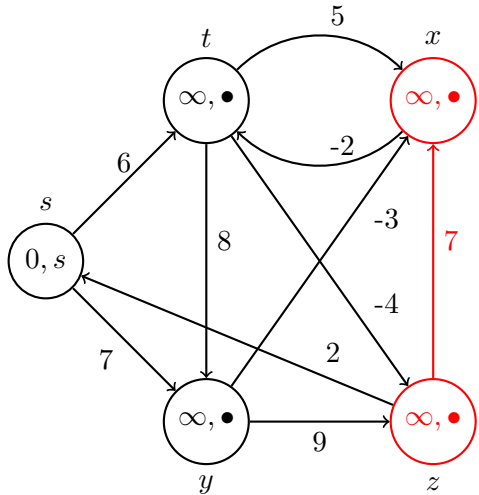
$i=1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	×	×					

Example: Bellman-Ford



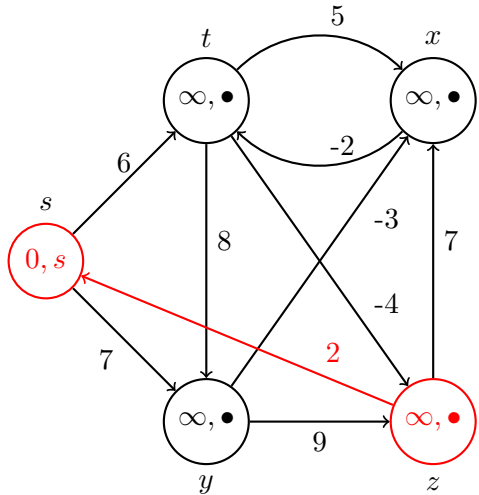
$i=1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	×	×	×				

Example: Bellman-Ford



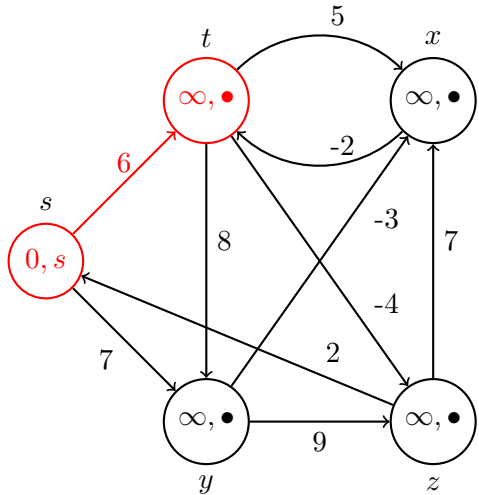
$i = 1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	×	×	×	×			

Example: Bellman-Ford



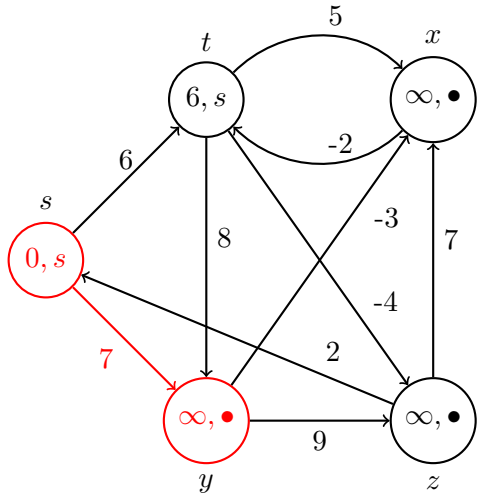
$i=1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	$(\boldsymbol{z,s})$	(s,t)	(s,y)
	×	×	×	×	×	×	×	×		

Example: Bellman-Ford



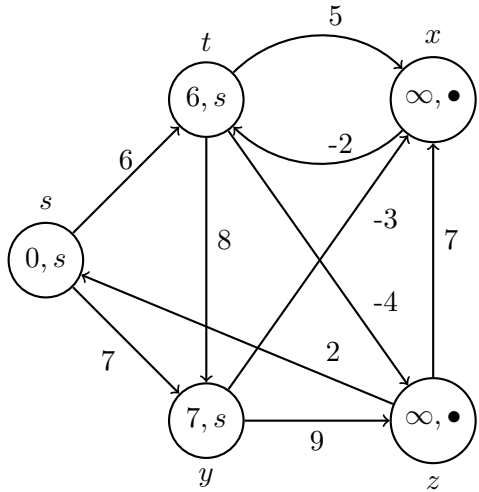
$i = 1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	$(\boldsymbol{s,t})$	(s,y)
	×	×	×	×	×	×	×	×	✓	

Example: Bellman-Ford



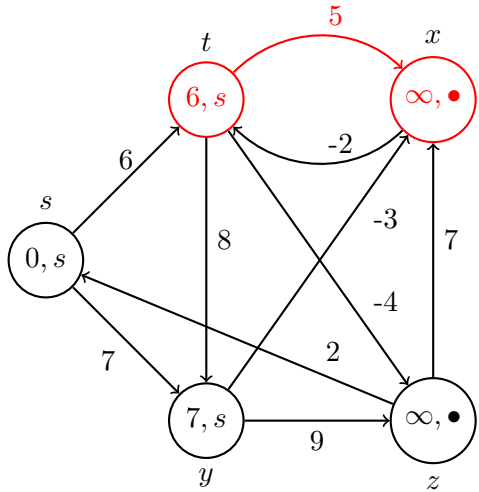
$i=1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	$\boldsymbol{(s,y)}$
	×	×	×	×	×	×	×	×	✓	✓

Example: Bellman-Ford



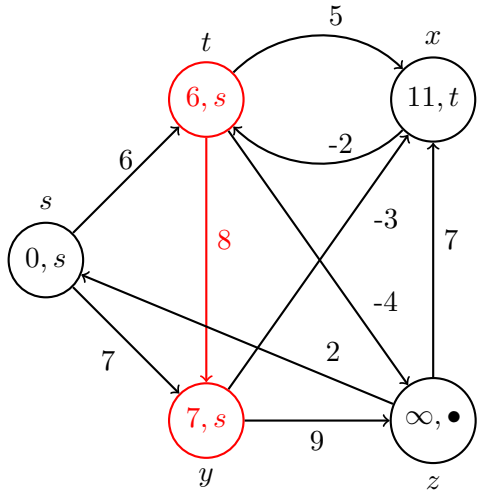
$i=1$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	×	×	×	×	×	✓	✓

Example: Bellman-Ford



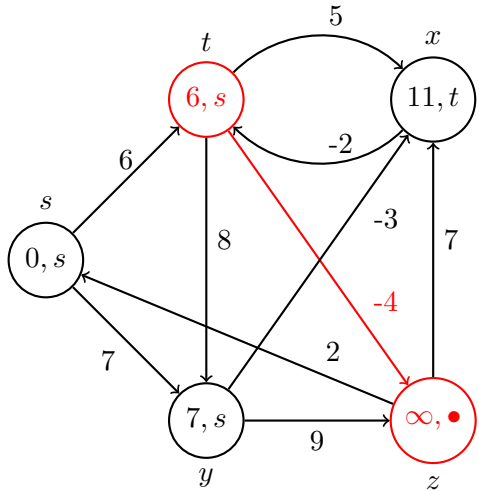
$i = 2$	(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y, z)	(z, x)	(z, s)	(s, t)	(s, y)
	✓									

Example: Bellman-Ford



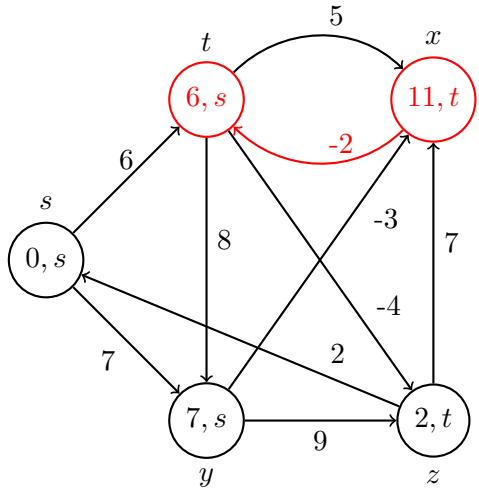
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	✓	×								

Example: Bellman-Ford



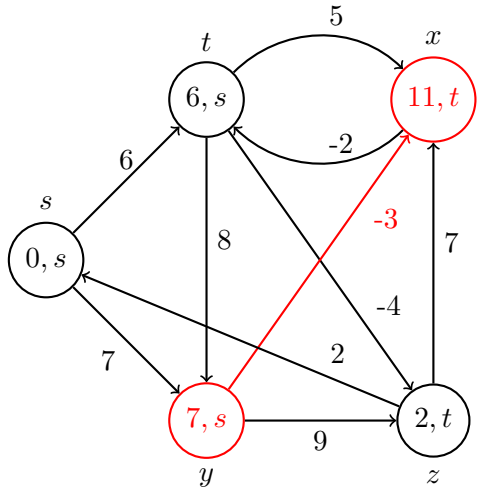
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	✓	×	✓							

Example: Bellman-Ford



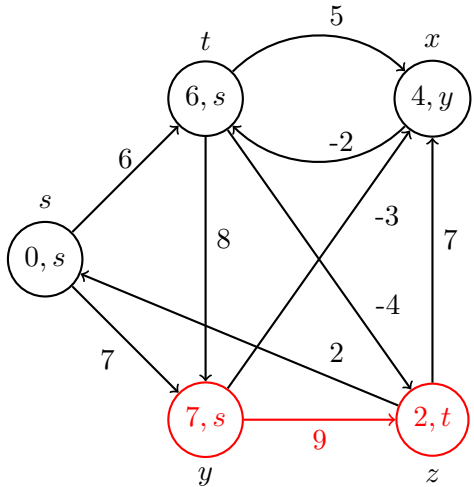
$i = 2$	(t,x)	(t,y)	(t,z)	$(\boldsymbol{x},\boldsymbol{t})$	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	✓	×	✓	×						

Example: Bellman-Ford



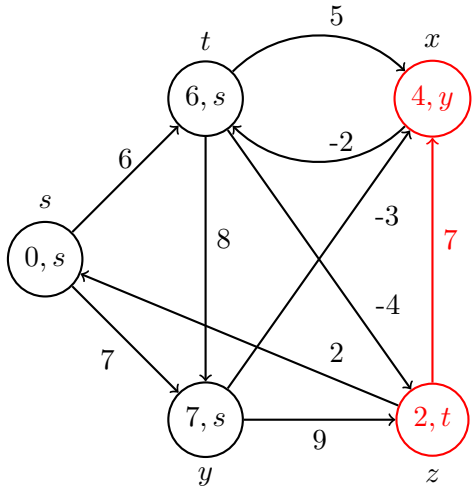
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	$(\boldsymbol{y},\boldsymbol{x})$	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	✓	×	✓	×	✓					

Example: Bellman-Ford



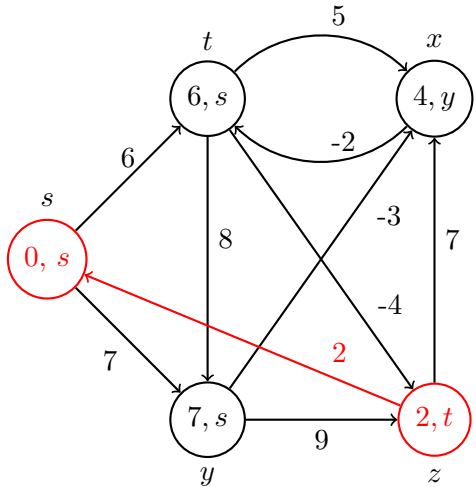
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	✓	×	✓	×	✓	×				

Example: Bellman-Ford



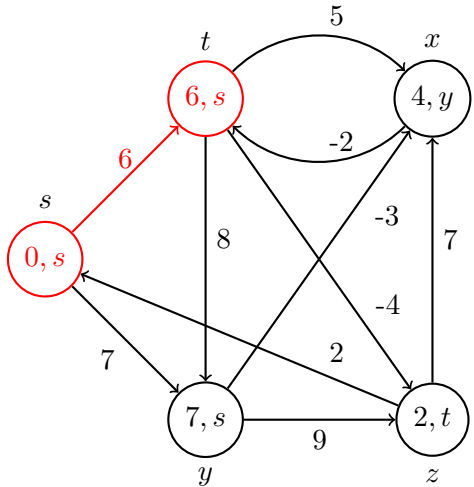
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	$(\boldsymbol{z},\boldsymbol{x})$	(z,s)	(s,t)	(s,y)
	✓	×	✓	×	✓	×	×			

Example: Bellman-Ford



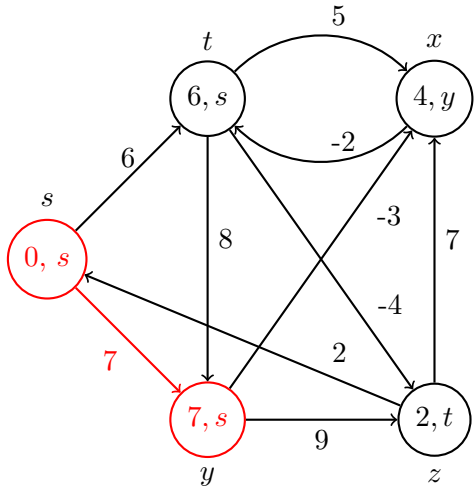
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	✓	×	✓	×	✓	×	×	×		

Example: Bellman-Ford



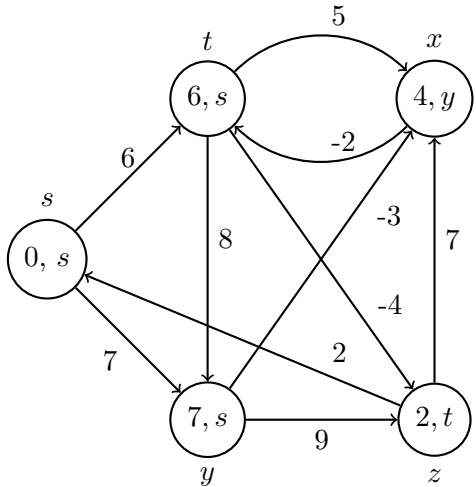
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	$(\boldsymbol{s,t})$	(s,y)
	✓	×	✓	×	✓	×	×	×	×	

Example: Bellman-Ford



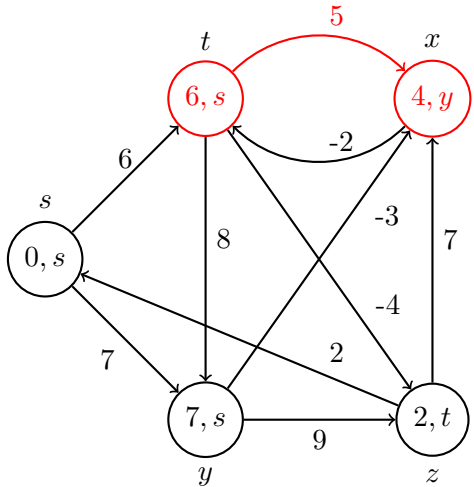
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	$\boldsymbol{(s,y)}$
	✓	×	✓	×	✓	×	×	×	×	×

Example: Bellman-Ford



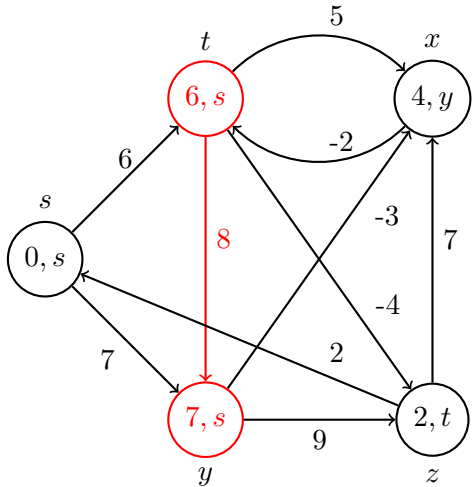
$i = 2$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	✓	×	✓	×	✓	×	×	×	×	×

Example: Bellman-Ford



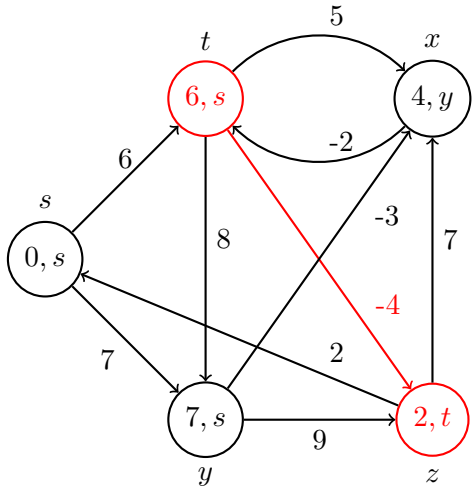
$i = 3$	(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y, z)	(z, x)	(z, s)	(s, t)	(s, y)
	×									

Example: Bellman-Ford



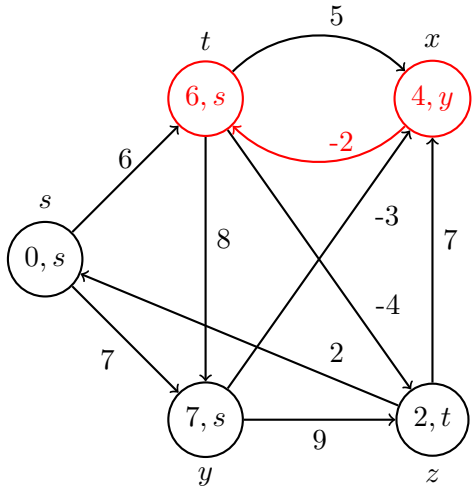
$i = 3$	(t,x)	$\boldsymbol{(t,y)}$	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×								

Example: Bellman-Ford



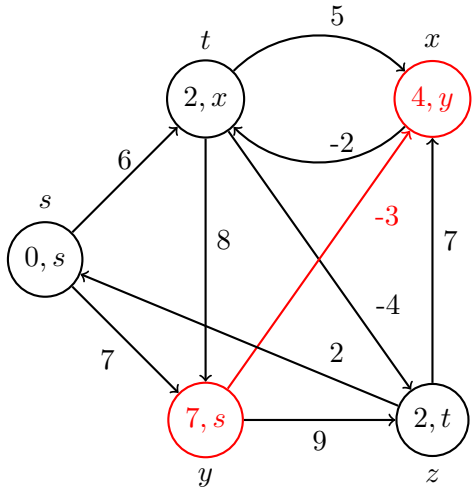
$i = 3$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×							

Example: Bellman-Ford



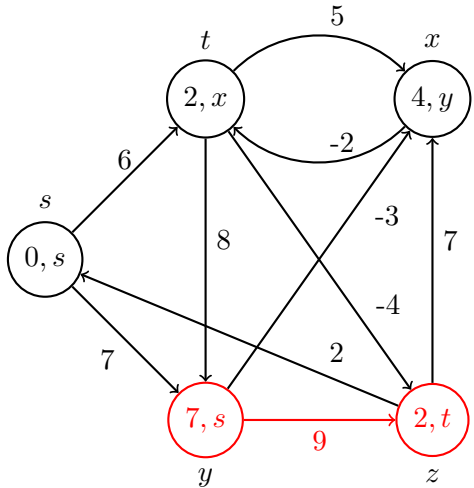
$i = 3$	(t,x)	(t,y)	(t,z)	$(\boldsymbol{x},\boldsymbol{t})$	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	✓						

Example: Bellman-Ford



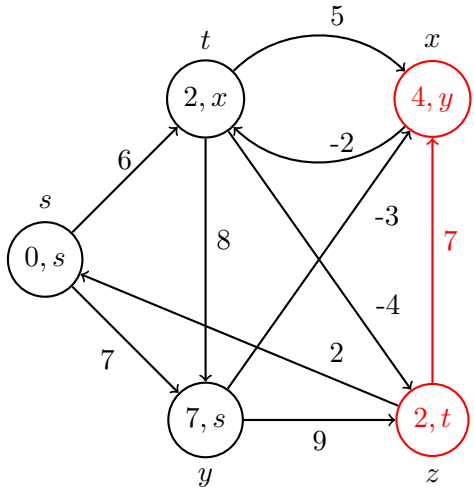
$i = 3$	(t,x)	(t,y)	(t,z)	(x,t)	$(\boldsymbol{y},\boldsymbol{x})$	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	✓	×					

Example: Bellman-Ford



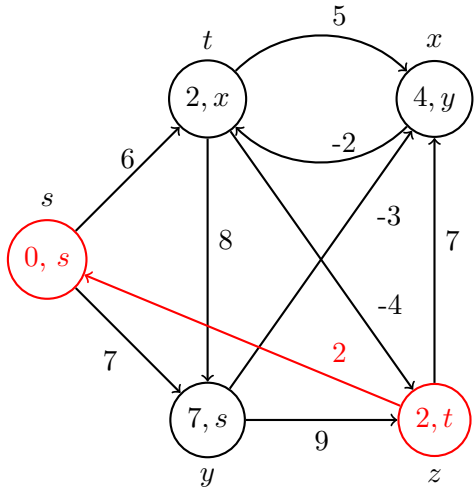
$i = 3$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	✓	×	×				

Example: Bellman-Ford



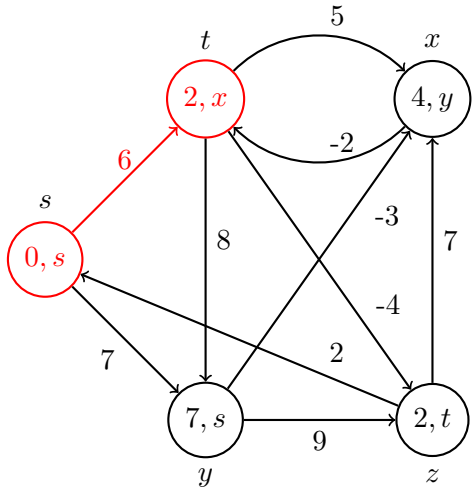
$i = 3$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	$(\boldsymbol{z},\boldsymbol{x})$	(z,s)	(s,t)	(s,y)
	×	×	×	✓	×	×	×			

Example: Bellman-Ford



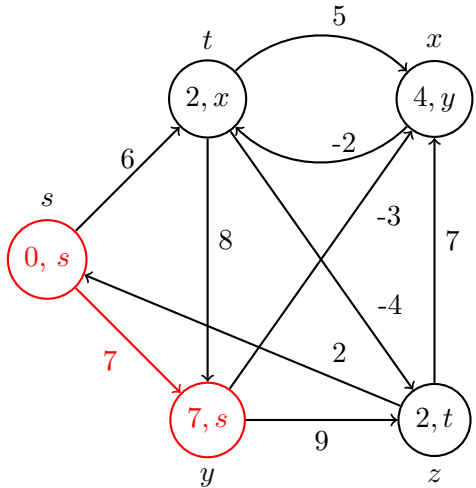
$i = 3$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	✓	×	×	×	×		

Example: Bellman-Ford



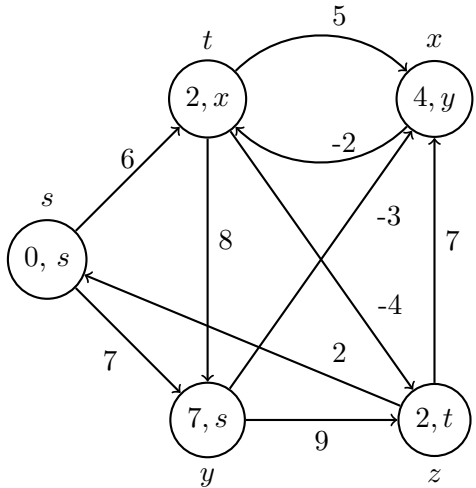
$i = 3$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	$(\boldsymbol{s,t})$	(s,y)
	×	×	×	✓	×	×	×	×	×	

Example: Bellman-Ford



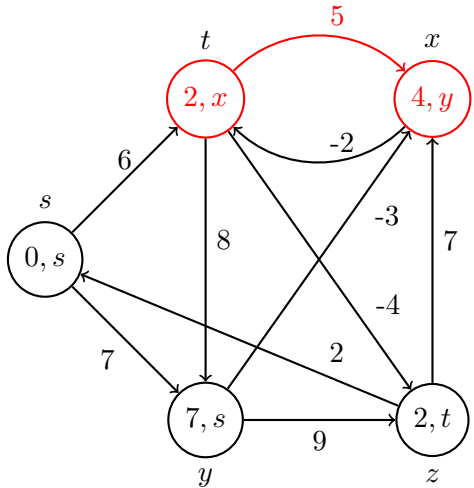
$i = 3$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	$\boldsymbol{(s,y)}$
	×	×	×	✓	×	×	×	×	×	×

Example: Bellman-Ford



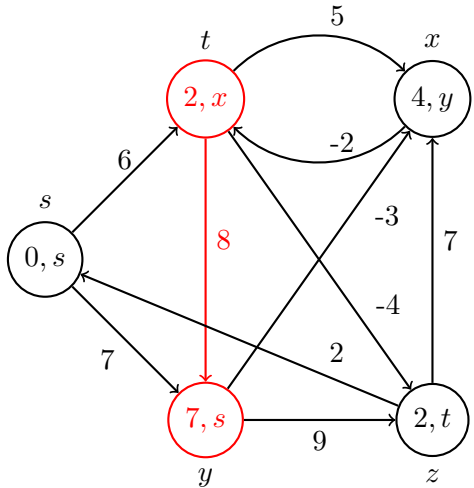
$i = 3$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	$\times$	$\times$	$\times$	✓	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Example: Bellman-Ford



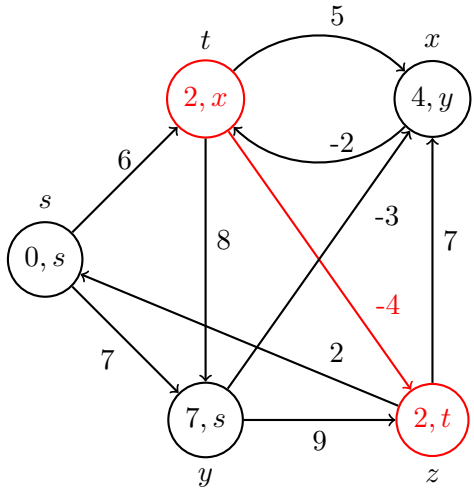
$i = 4$	(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y, z)	(z, x)	(z, s)	(s, t)	(s, y)
	×									

Example: Bellman-Ford



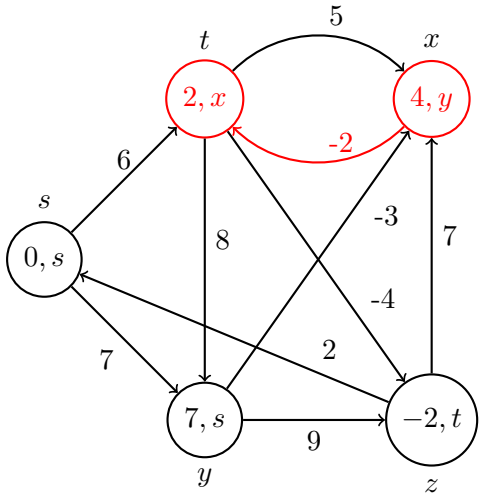
$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×								

Example: Bellman-Ford



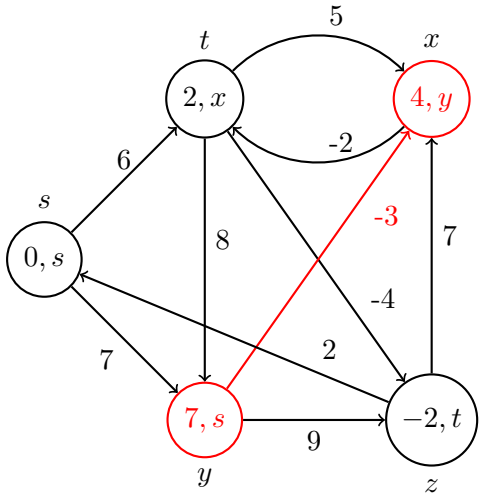
$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	✓							

Example: Bellman-Ford



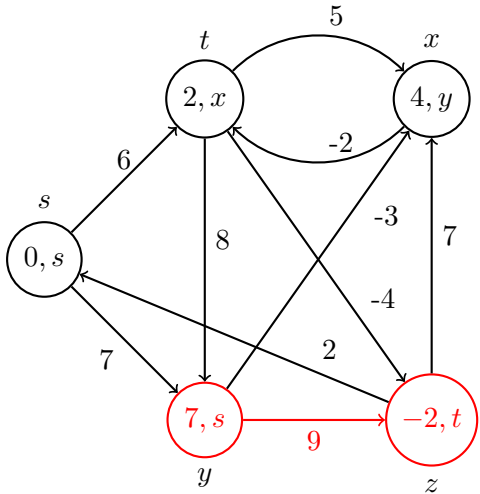
$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	✓	×						

Example: Bellman-Ford



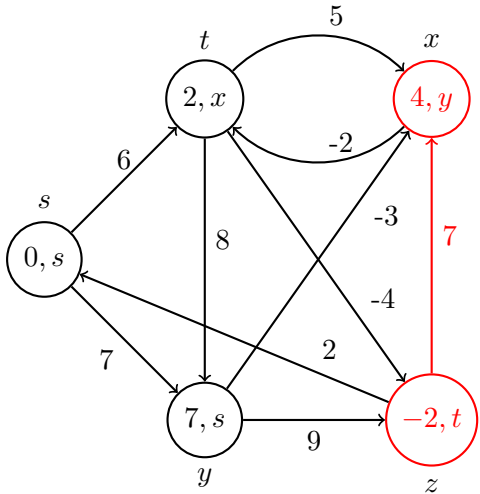
$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	$(\boldsymbol{y},\boldsymbol{x})$	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	✓	×	×					

Example: Bellman-Ford



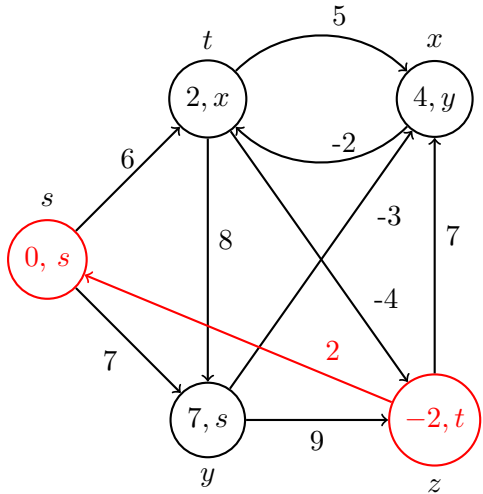
$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	✓	×	×	×				

Example: Bellman-Ford



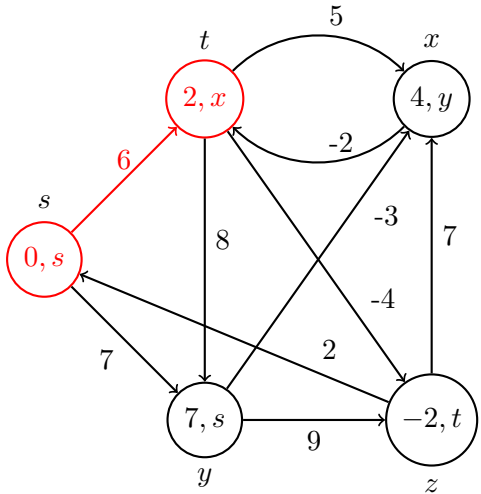
$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	✓	×	×	×	×			

Example: Bellman-Ford



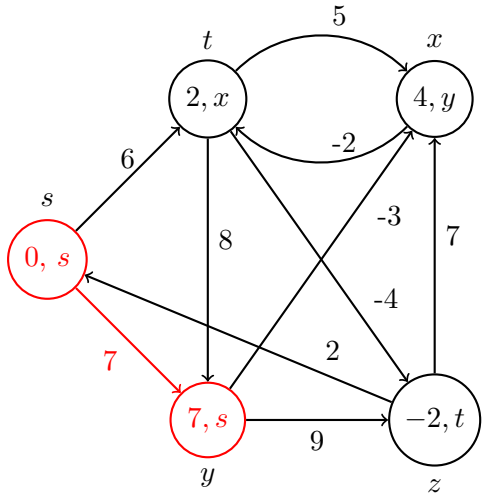
$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	✓	×	×	×	×	×		

Example: Bellman-Ford



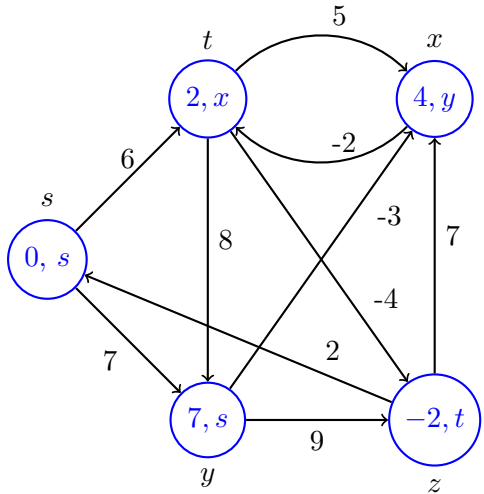
$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	$(\boldsymbol{s,t})$	(s,y)
	×	×	✓	×	×	×	×	×	×	

Example: Bellman-Ford



$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	✓	×	×	×	×	×	×	×

Example: Bellman-Ford



$i = 4$	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)
	×	×	×	✓	×	×	×	×	×	×

The Floyd-Warshall algorithm

Outlook

Floyd-Warshall

- **no fixed source**: computes all distances $\delta(u, v)$
- negative weights OK but **no negative cycle**
  (can be tested in $\Theta(mn)$ with Bellman-Ford)
- very simple pseudo-code, but slower than other algorithms
- another application of dynamic programming

Remark: doing Bellman-Ford from all u takes $\boldsymbol{\Theta(mn^2)}$

Looking at subsets of vertices

Subproblems for dynamic programming

- Bellman-Ford uses paths with **fixed numbers of steps**
- Floyd-Warshall restricts which **vertices** can be used

Definition:

- for $i = 0, \ldots, n$, set $\boldsymbol{D_i(v_j, v_k)}$ = length of the shortest walk $v_j \rightsquigarrow v_k$ with all intermediate vertices in $v_1, \ldots, v_i$
- for $i = 0$, we get
  - $D_0(v_j, v_j) = 0$
  - $D_0(v_j, v_k) = w(v_j, v_k)$ if there is an edge (v_j, v_k)
  - $D_0(v_j, v_k) = \infty$ otherwise
- $D_n(v_j, v_k) = \delta(v_j, v_k)$

Pseudo-code

Claim

$D_i(v_j, v_k) = \min(\boldsymbol{D_{i-1}(v_j, v_k)}, \boldsymbol{D_{i-1}(v_j, v_i) + D_{i-1}(v_i, v_k)})$

Proof: either the shortest path does not go through v_i, or it does (if it does, it's only once)

FloydWarshall(G)

1. set up D_0 above
2. **for** $i = 1, \ldots, n$ **do**
3. **for** $j = 1, \ldots, n$ **do**
4. **for** $k = 1, \ldots, n$ **do**
5. $D_i[v_j, v_k] \leftarrow \min(D_{i-1}[v_j, v_k], D_{i-1}[v_j, v_i] + D_{i-1}[v_i, v_k])$

Analysis

Runtime and memory: $\Theta(n^3)$

Exercise 1

prove that we can use only a single array $D[v_j, v_k]$, with

$$D[v_j, v_k] \leftarrow \min(D[v_j, v_k], D[v_j, v_i] + D[v_i, v_k])$$

Hint: if no negative cycle, this computes the same values (unlike Bellman-Ford)

Exercise 2

to find all shortest paths, use an array $P[v_j, v_k]$, which gives the vertex following v_j on the shortest path to v_k