Foundations of Algorithms

Dynamic Programming: Introduction & Weighted Interval Scheduling

Ira A. Fulton Schools of Engineering
Arizona State University

Introduction & Weighted Interval Scheduling

Objectives

- Introduce Dynamic Programming and its applications
- Explain solution of the Weighted Interval Scheduling problem using Dynamic Programming

Algorithmic Paradigms

Greedy

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

Divide-and-conquer

- Break up a problem into **disjoint sub-problems**, solve each sub-problem independently, 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.

Dynamic Programming Applications

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

Application areas

- Bioinformatics
- Operations research
- Control theory
- Information theory
- Computer graphics
- Artificial intelligence

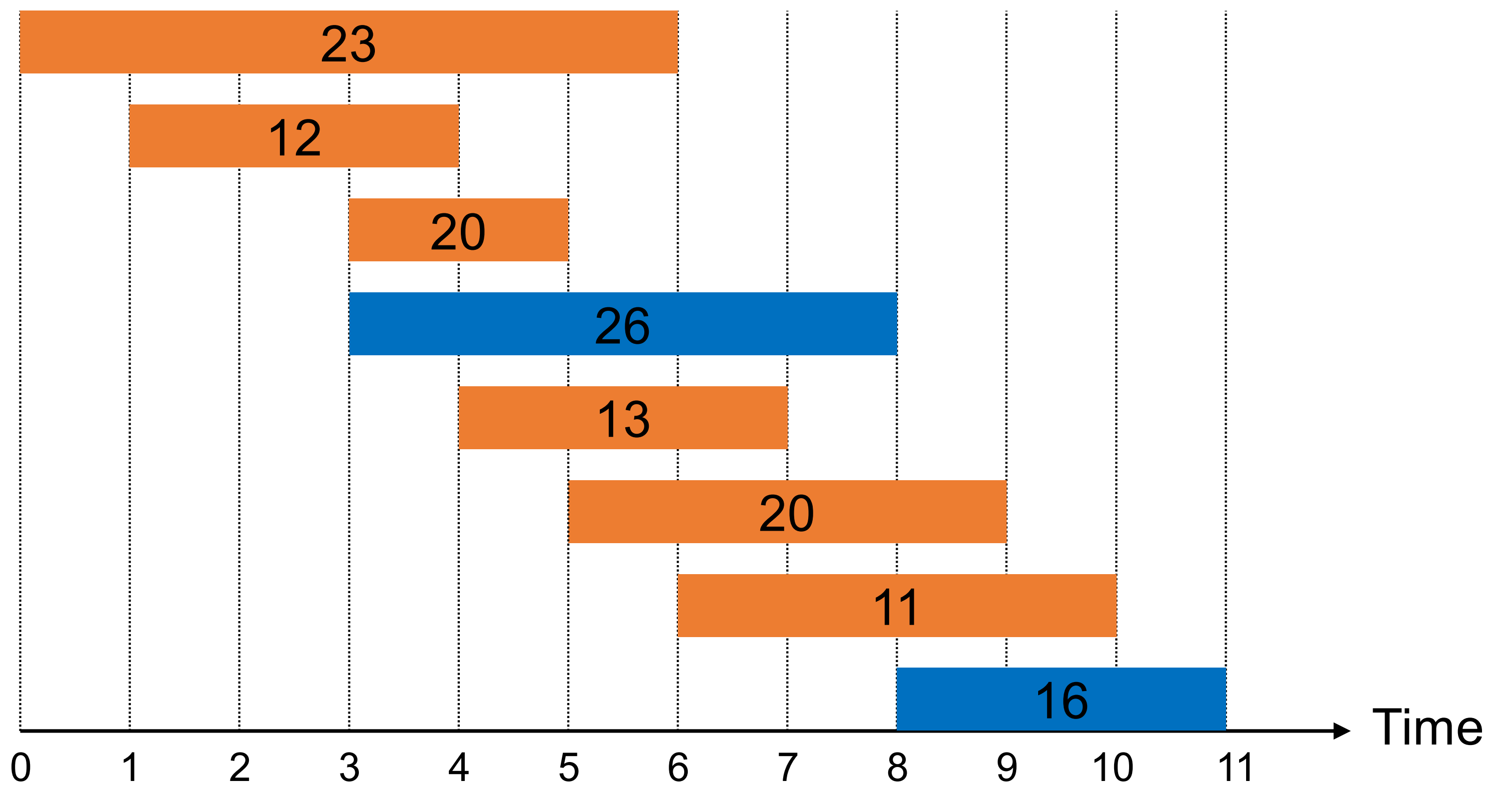
Some famous dynamic programming algorithms.

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

Weighted Interval Scheduling

Weighted interval scheduling problem

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

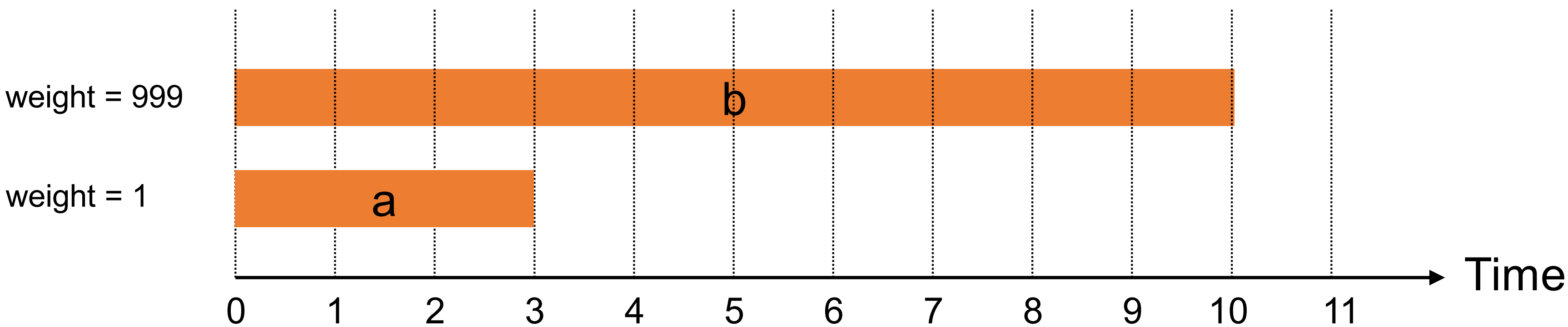


Unweighted Interval Scheduling Review

Recall that greedy algorithm works if all weights are 1.

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

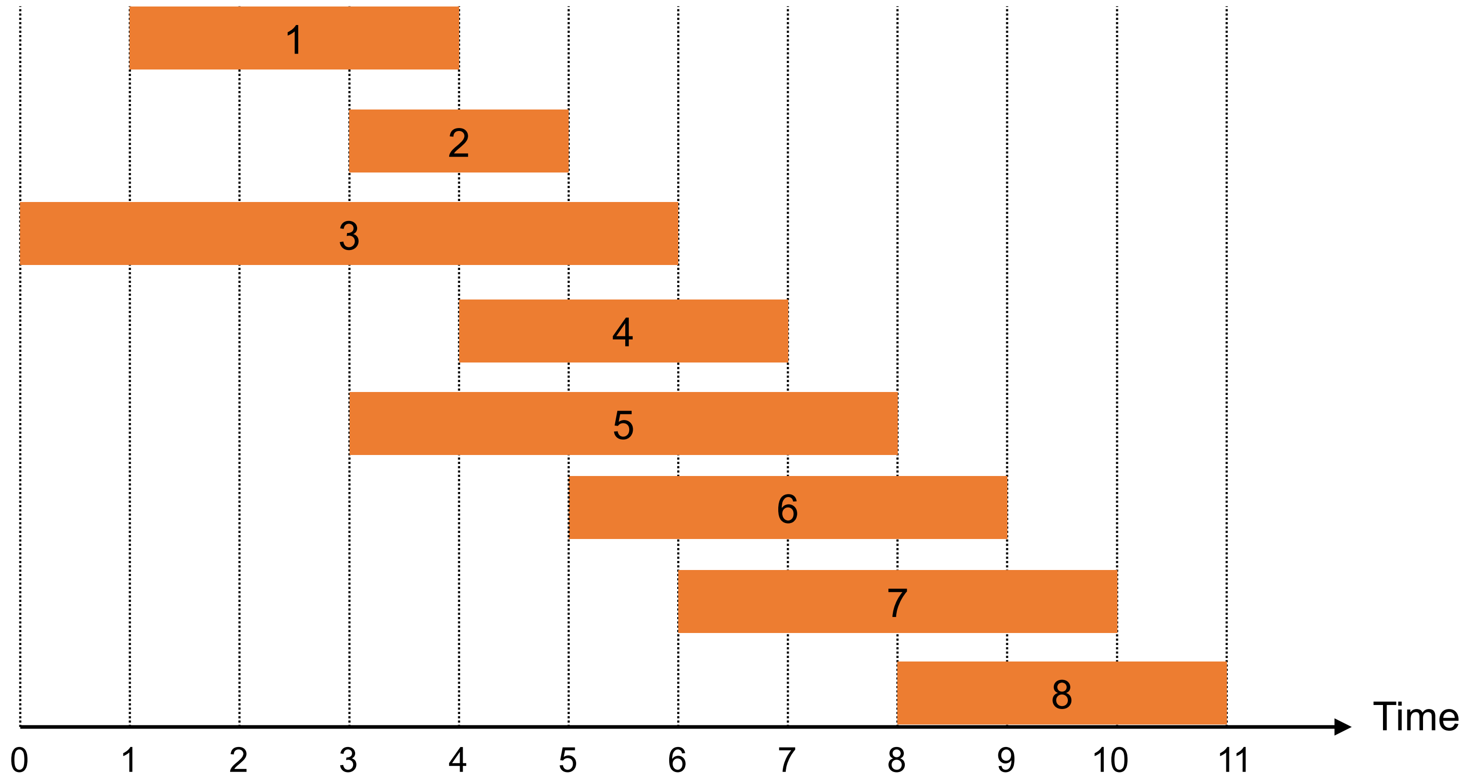
Observation. Greedy algorithm can fail if arbitrary weights are allowed.



Weighted Interval Scheduling

- Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Define p(j) = largest index i < j such that job i is compatible with j.

- Example: p(8) = 5, p(7) = 3, p(2) = 0.



Dynamic Programming: Binary Choice

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

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to the problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to the problem consisting of remaining compatible jobs 1, 2, ..., j-1

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

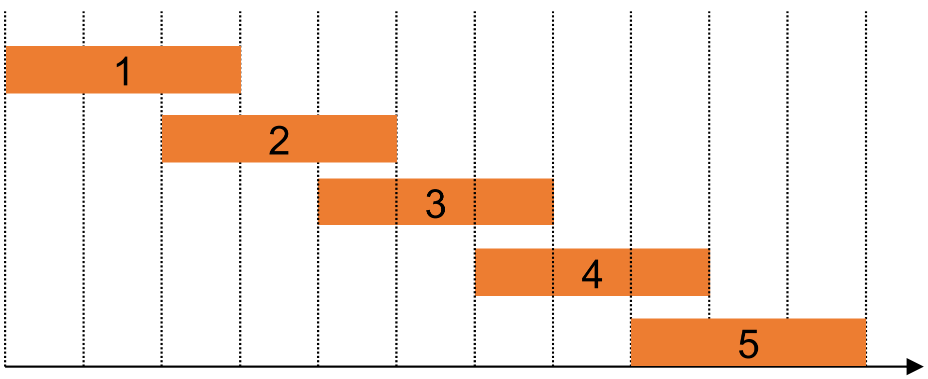
Weighted Interval Scheduling: Brute Force

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

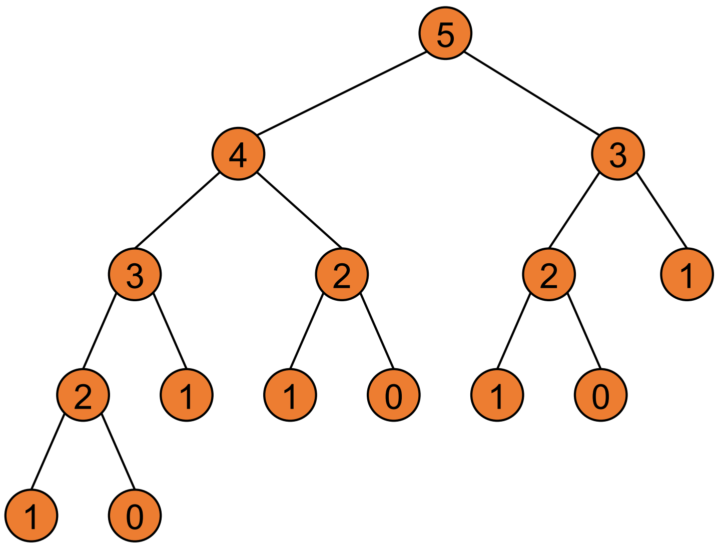
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails due to redundant sub-problems $\Rightarrow$ exponential algorithms.

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



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache and lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

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

M-compute(n)

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + 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(·) : O(n) after sorting by start time (which takes O(n log n) time).

- `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 Φ = # 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 `M-Compute-Opt(n)` is O(n).

Remark. Running time is O(n) if jobs are pre-sorted by start and finish times.

Automated Memoization

- Many functional programming languages (e.g., Lisp) have built-in support for memoization.

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

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

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

Summary

Foundations of Algorithms

Knapsack Problem

Ira A. Fulton Schools of Engineering
Arizona State University

Knapsack Problem

Objectives

- **Explain solution of the Knapsack problem using Dynamic Programming**

Knapsack Problem

Knapsack problem.

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

Example: { 3, 4 } has value 40.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy algorithm: repeatedly add item with maximum ratio v_i / w_i.

Example: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy algorithm is not optimal.

Dynamic Programming: False Start

- Define OPT(i) = max profit subset of items 1, …, i.

  - Case 1: OPT does not select item i.
    - OPT selects best of { 1, 2, …, i-1 }.

  - Case 2: OPT selects item i.
    - accepting item i does not imply rejecting other items.
    - Without knowing what other problems were selected before item I, we do not even know if there is enough room for item i.

- Conclusion. We need more sub-problems!

Dynamic Programming: Adding a New Variable

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

  - Case 1: OPT does not select item i.
    - OPT selects best of { 1, 2, …, i-1 } using weight limit w.

  - Case 2: OPT selects item i.
    - new weight limit = $w - w_i$.
    - OPT selects best of { 1, 2, …, i-1 } using this new weight limit.

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

Knapsack Problem: Bottom-Up

- The Knapsack problem can be considered as filling up an n-by-W array using the following algorithm.

```
Input: n, W, w1,…,wn, v1,…,vn

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}

return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	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: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

Running time. Θ(n W).

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm.

- There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.

Summary

Foundations of Algorithms

Shortest Paths: Bellman-Ford

Ira A. Fulton Schools of Engineering
Arizona State University

Shortest Paths

Objectives

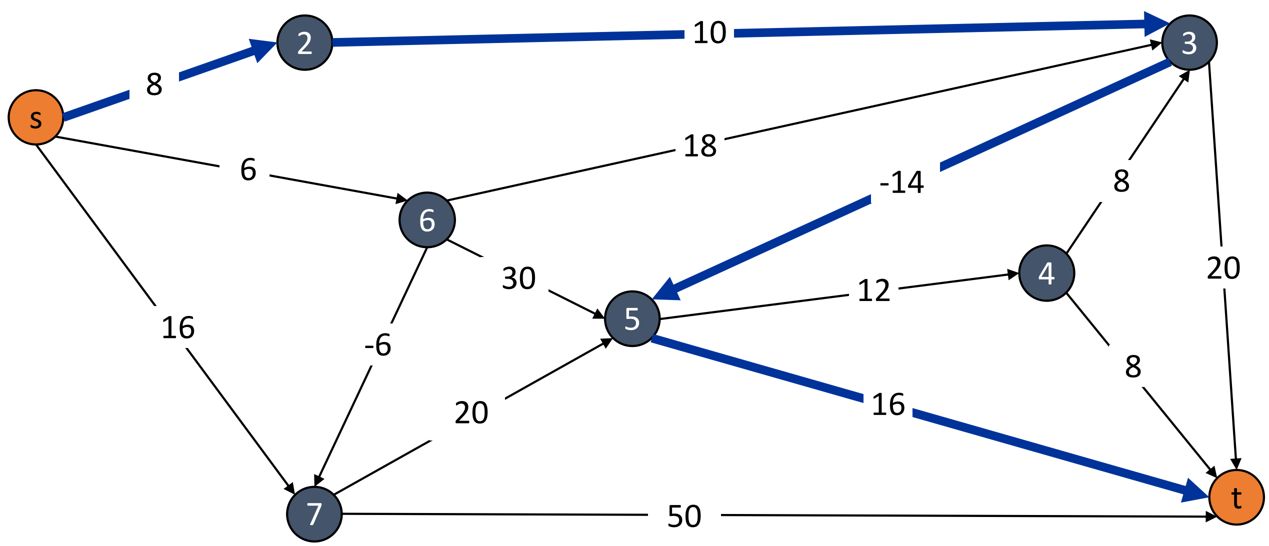
- **Explain solution of the Shortest Path problem using Dynamic Programming and Bellman-Ford algorithm**

Shortest Paths

Shortest path problem. Given a directed graph G = (V, E), with edge weights c_{vw} , find the shortest path from node s to t.

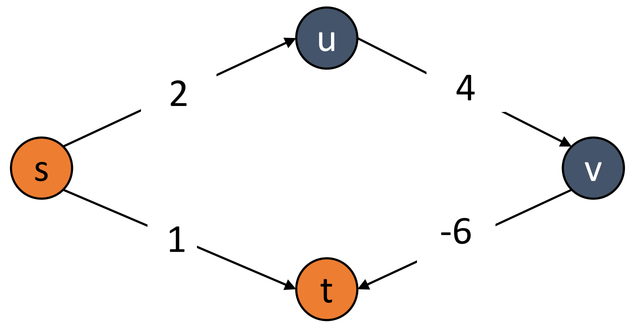
allow negative weights

Example. In the following graph, nodes are agents in a financial setting and c_{vw} is transaction cost when we buy from agent v and sell to w. What is the total minimum cost from node s to t?

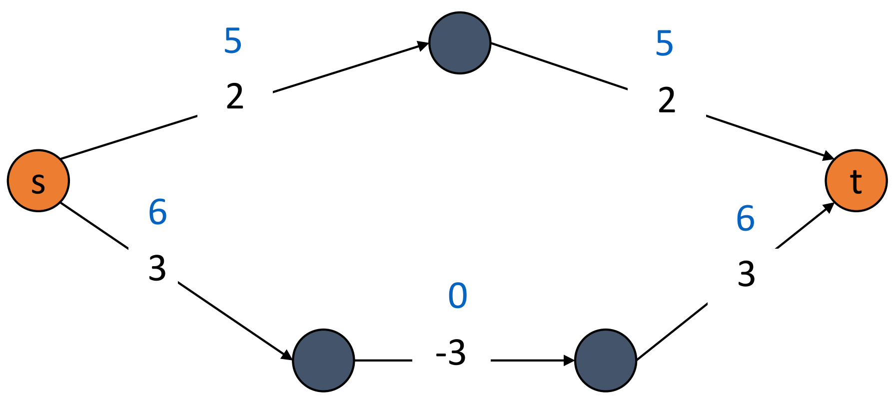


Shortest Paths: Failed Attempts

Dijkstra's algorithm can fail if there are negative edge costs.

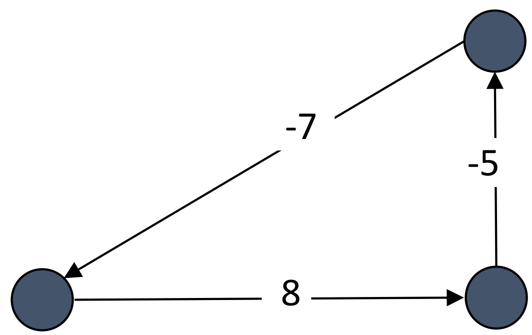


Re-weighting. Adding a constant to every edge weight can change the solution.

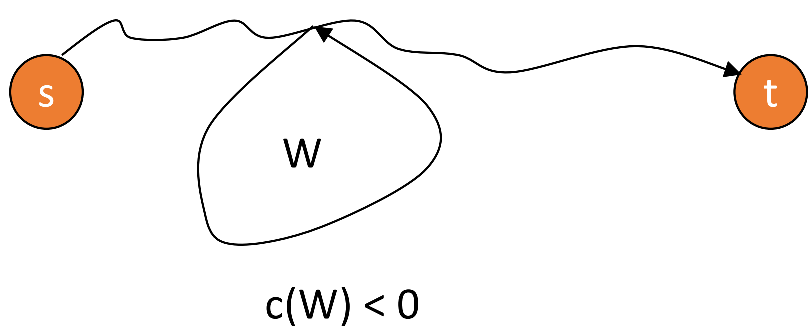


Shortest Paths: Negative Cost Cycles

Negative cost cycle.



Observation. 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.



Shortest Paths: Dynamic Programming

Definition. 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 the first edge, then OPT uses (v, w) and then selects best w-t path using at most i-1 edges

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

Remark. By previous observation, if there are no negative cycles, then OPT(n-1, v) = length of shortest v-t path.

Shortest Paths: Implementation

```
Shortest-Path(G, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
      foreach edge (v, w) ∈ E
         M[i, v] ← min { M[i, v], M[i-1, w] + c_vw }
}
```

This algorithm computes the length of a shortest path in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Finding the shortest paths. Maintain a successor [i, v] that points to next node on a shortest v-t path using at most i edges.

Shortest Paths: Practical Improvements

Practical improvements.

- Maintain only one array M[v] = shortest v-t path that we have found so far.
- No need to check edges of the form (v, w) unless M[w] changed in previous iteration.

Theorem. Throughout the algorithm, M[v] is length of some v-t path, and after i rounds of updates, the value M[v] is no larger than the length of shortest v-t path using $\leq$ i edges.

Overall impact.

- Memory: O(m + n).
- Running time: O(mn) worst case, but substantially faster in practice.

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V {
      M[v] ← ∞
      successor[v] ← ϕ
   }

   M[t] = 0
   for i = 1 to n-1 {
      foreach node w ∈ V {
      if (M[w] has been updated in previous iteration) {
         foreach node v such that (v, w) ∈ E {
            if (M[v] > M[w] + c_vw) {
               M[v] ← M[w] + c_vw
               successor[v] ← w
            }
         }
      }
      If no M[w] value changed in iteration i, stop.
   }
}
```

Summary

Foundations of Algorithms

Distance Vector Protocol

Ira A. Fulton Schools of Engineering
Arizona State University

Distance Vector Protocol

Objectives

- **Explain Distance Vector Protocol**

Distance Vector Protocol

- We can apply the shortest path problem to routers in a communication network to determine the most efficient path to a destination.
  - nodes represent routers,
  - edges represent direct communication links,
  - cost of an edge is the delay on the link.

- We can use Dijkstra's algorithm to solve this problem but it requires global knowledge of the network.

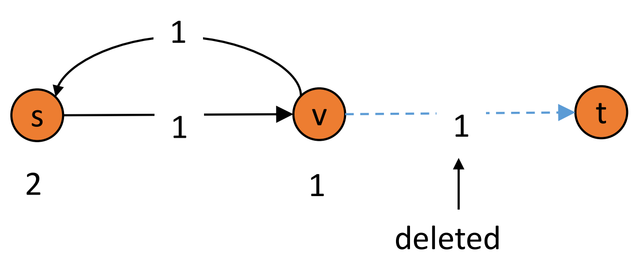
- The Bellman-Ford algorithm uses only local knowledge of neighboring nodes.

- We don't expect routers to run in lockstep. The order in which each `foreach` loop executes is not important. Moreover, algorithm still converges even if updates are asynchronous.

Distance Vector Protocol

- Distance vector protocol (Routing by rumor)
  - Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions).
  - Each router performs n separate computations, one for each potential destination node.
  - Examples: Routing Information Protocol (RIP), Xerox XNS RIP, Novell IPX RIP, Cisco IGRP, AppleTalk RTMP.

- Caveats
  - Edge costs may change during algorithm.
  - If edge (v, t) below is deleted, the Bellman-Ford algorithm will begin counting to infinity.



"counting to infinity"

Path Vector Protocols

- To avoid the problems of the Distance Vector Protocol, network designers adopted Path Vector Protocol:
  - Each router stores the entire path (not just the distance and first hop).
  - Based on Dijkstra's algorithm.
  - Requires significantly more storage.

- Examples using the Path Vector Protocol:
  - Border Gateway Protocol (BGP)
  - Open Shortest Path First (OSPF)

Summary

Foundations of Algorithms

Negative Cycles in a Graph

Ira A. Fulton Schools of Engineering
Arizona State University

Negative Cycles in a Graph

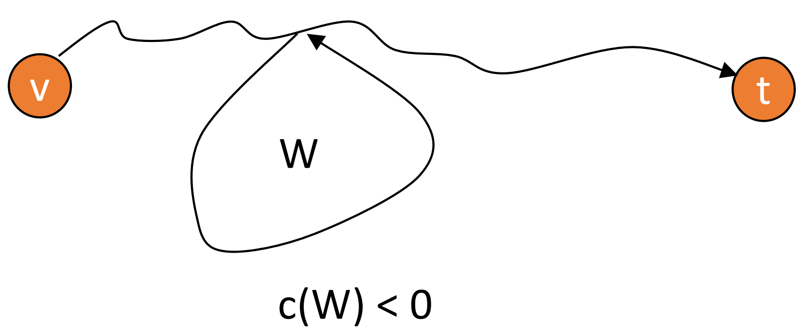
Objectives

- Explain how to detect negative cycles in a graph

Detecting Negative Cycles

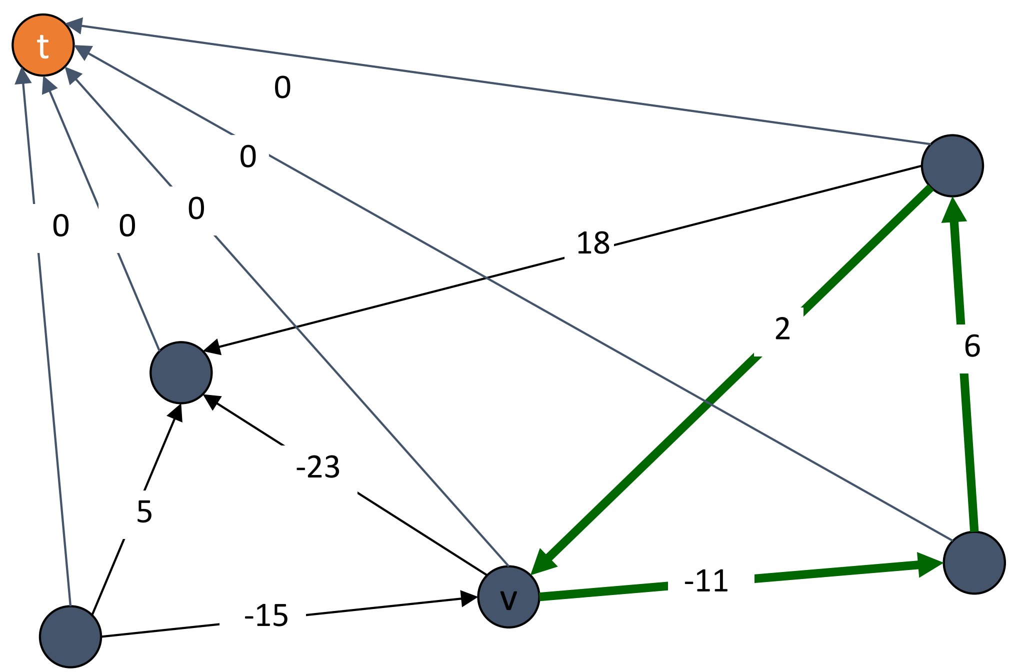
- Lemma. If OPT(n,v) = OPT(n-1,v) for all v, then there are no negative cycles on paths to node t.
- Proof. Bellman-Ford algorithm.

- Lemma. If OPT(n,v) < OPT(n-1,v) for some node v, then (any) shortest path from v to t contains a cycle W. Moreover W has negative cost.
- Proof.
  - Since OPT(n,v) < OPT(n-1,v), we know P has exactly n edges.
  - By pigeonhole principle, P must contain a directed cycle W.
  - Deleting W yields a v-t path with < n edges $\Rightarrow$ W has negative cost.



Detecting Negative Cycles

- Theorem. One can detect negative cost cycle in O(mn) time.
  - Add new node t and connect all nodes to t with 0-cost edge.
  - Check if OPT(n, v) = OPT(n-1, v) for all nodes v.
    - if yes, then there are no negative cycles
    - if no, then extract cycle from shortest path from v to t



Detecting Negative Cycles: Application

- Currency conversion. Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?