Dynamic Programming

Data Structures and Algorithms

Andrei Bulatov

Weighted Interval Scheduling

Weighted interval scheduling problem.

Instance

A set of n jobs.

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.

Objective

Find maximum weight subset of mutually compatible jobs.

Unweighted Interval Scheduling: Review

Recall:

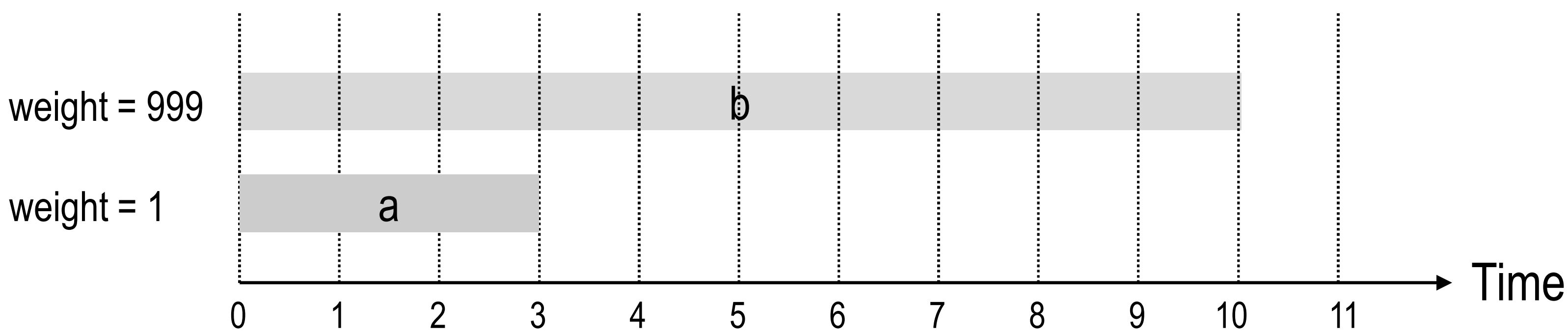
Greedy algorithm works if all weights are 1.

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 spectacularly if arbitrary weights are allowed.



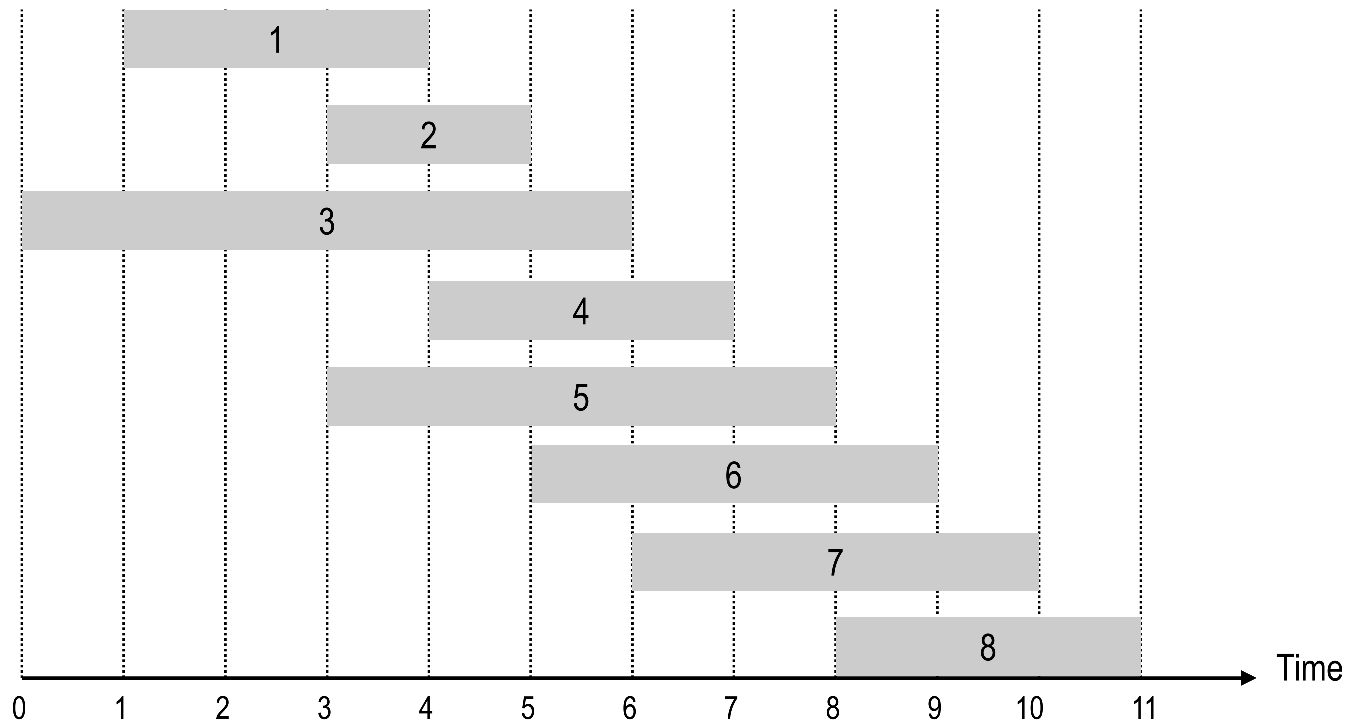
Weighted Interval Scheduling

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

Let $p(j)$ be the 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

Let OPT(j) denote the value of an optimal solution to the problem consisting of job requests 1, 2, ..., j.

Case 1: OPT selects job j.

cannot 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

Case 2: OPT does not select job j.

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

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

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)
  return Compute-Opt(n)


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

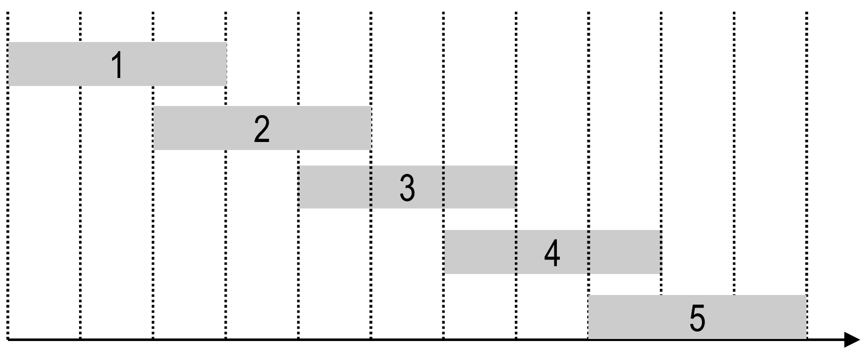
Weighted Interval Scheduling: Brute Force

Observation.

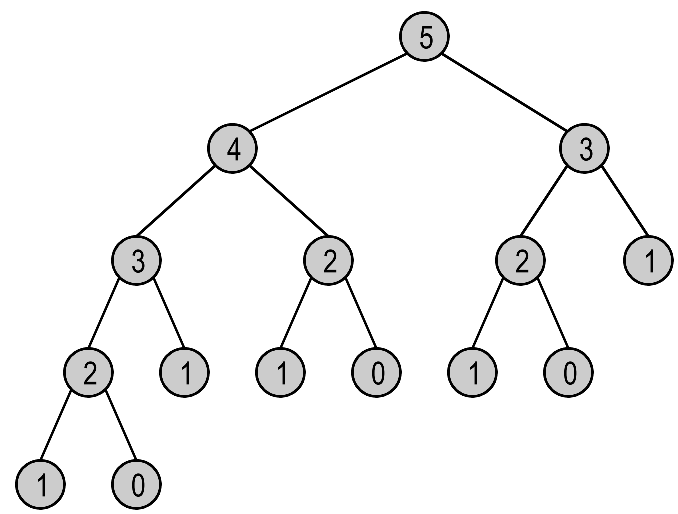
Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

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; lookup as needed.

```
Input:
   n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

sort jobs by finish times so that f_1≤f_2≤…≤ f_n
compute p(1), p(2), …, p(n)
set OPT[0]:=0
for j=1 to n do
   set OPT[j]:=max(v_j+OPT[p(j)],OPT[j-1])
endfor
return OPT[n]
```

Weighted Interval Scheduling: Running Time

Theorem

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

Proof

Sort by finish time: O(n log n).

Computing p(·) : O(n) after sorting by finish time

Each iteration of the for loop: O(1)

Overall time is O(n log n)

QED

Remark.

O(n) if jobs are pre-sorted by finish times

Automated Memoization

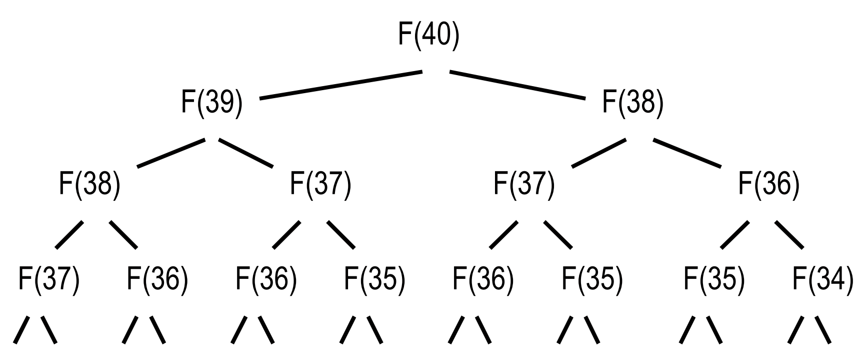
Automated memoization.
Many functional programming languages (e.g., Lisp) have built-in support for memoization.
Q. Why not in imperative languages (e.g., Java)?

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficient)

```
static int F(int n) {
   if (n <= 1) return n;
   else return F(n-1) + F(n-2);
}
```

Java (exponential)



Finding a Solution

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

Do some post-processing

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

Knapsack

The Knapsack Problem

Instance:

A set of n objects, each of which has a positive integer value v_i and a positive integer weight w_i. A weight limit W.

Objective:

Select objects so that their total weight does not exceed W, and they have maximal total value

Idea

A simple question: Should we include the last object into selection?

Let OPT(n,W) denote the maximal value of a selection of objects out of {1, …, n} such that the total weight of the selection doesn't exceed W

More general, OPT(i,U) denote the maximal value of a selection of objects out of {1, …, i} such that the total weight of the selection doesn't exceed U

Then

$$OPT(n,W) = \max\{ OPT(n-1, W),\ OPT(n-1, W - w_i) + v_i \}$$

Algorithm (First Try)

Knapsack(n,W)

```
set V1:=Knapsack(n-1,W)
set V2:=Knapsack(n-1,W- w_i )
output max(V1,V2+v_i )
```

Is it good enough?

Example

Let the values be 1,3,4,2, the weights 1,1,3,2, and W = 5

Recursion tree

Another Idea: Memoization

Let us store values OPT(i,U) as we find them

We need to store (and compute) at most $n \times W$ numbers

We'll do it in a regular way:

Instead of recursion, we will compute those values starting from smaller ones, and fill up a table

Algorithm (Second Try)

Knapsack(n,W)

```
array  M[0..n,0..W]
set M[0,w]:=0 for each w=0,1,...,W
for i=1 to n do
   for w=0 to W do
      set M[i,w]:= max{M[i-1,w],M[n-1,w-w_i]+v_i}
   endfor
endfor
```

Example

Example

Let the values be 1,3,4,2, the weights 1,1,3,2, and W = 5

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	1	3	3	3
2	0	1	4	4	4
3	0	1	4	4	5
4	0	1	4	7	7
5	0	1	4	8	8

M[i,w] = max{ M[i – 1, w], M[n – 1,w – w_i] + v_i }

Shortest Path

Suppose that every arc e of a digraph G has length
(or cost, or weight, or …) len(e)
But now we allow negative lengths (weights)

Then we can naturally define the length of a directed path in G,
and the distance between any two nodes

The s-t-Shortest Path Problem

Instance:

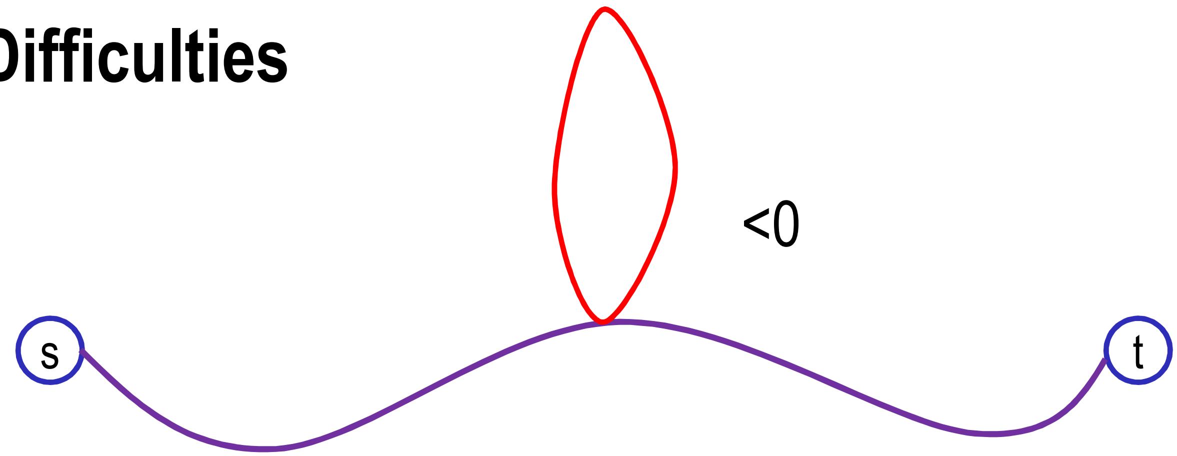
Digraph G with lengths of arcs, and nodes s,t

Objective:

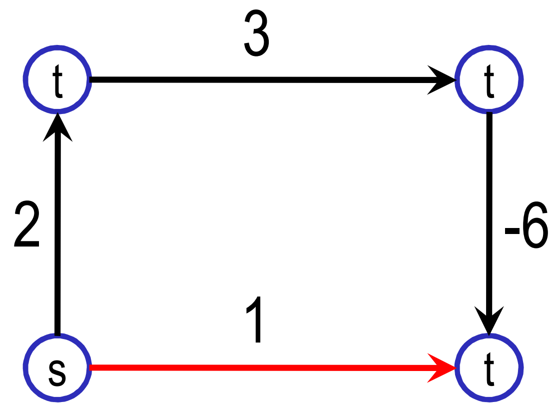
Find a shortest path between s and t

Shortest Path: Difficulties

Negative Cycles.



Greediness fails



Adding constant weight to all arcs fails

Shortest Path: Observations

Assumption

There are no negative cycles

Lemma

If graph G has no negative cycles, then there is a shortest path from s to t that is simple (i.e. does not repeat nodes), and hence has at most $n - 1$ arcs

Proof

If a shortest path P from s to t repeat a node v, then it also include a cycle C starting and ending at v.

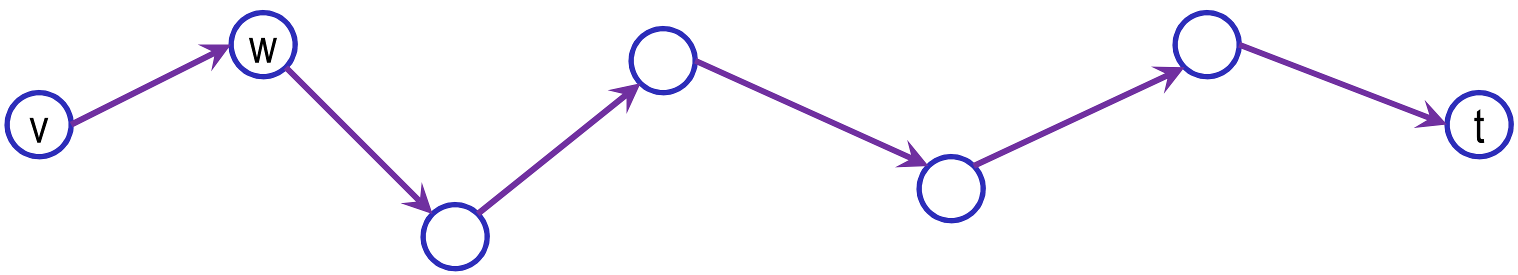
The weight of the cycle is non-negative, therefore removing the cycle makes the path shorter (no longer).

QED

Shortest Path: Dynamic Programming

We will be looking for a shortest path with increasing number of arcs

Let OPT(i,v) denote the minimum weight of a path from v to t using at most i arcs



Shortest v – t path can use i – 1 arcs. Then OPT(i,v) = OPT(i – 1,v)

Or it can use i arcs and the first arc is vw. Then

OPT(i,v) = len(vw) + OPT(i – 1,w)

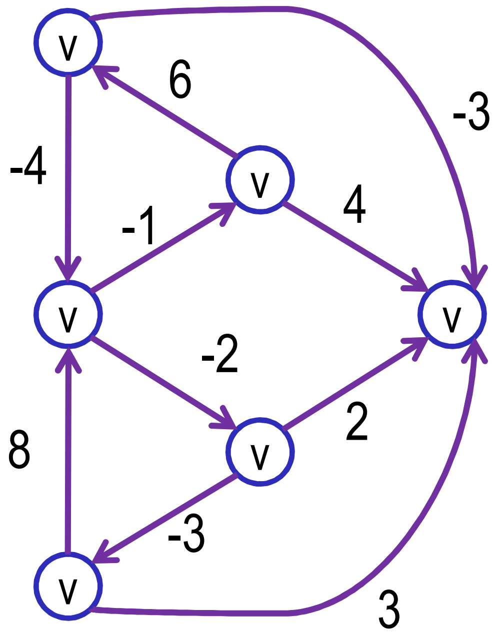
$$OPT(i,v) = \min\{OPT(i-1,v), \min_{w \in V}\{OPT(i-1,w) + len(vw)\}\}$$

Shortest Path: Bellman-Ford Algorithm

Shortest-Path(G,s,t)

```
set n:=|V|              /*number of nodes in G
array  M[0..n-1,V]
set M[0,t]:=0 and M[0,v]:=∞ for each v∈V-{t}
for i=1 to n-1 do
   for v∈V do
    set M[i,w]:=min{M[i-1,v],min_{w∈V}{M[i-1,w]+len(vw)}}
   endfor
endfor
return m[n-1,s]
```

Example



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

$$M[i,w] = \min\{ M[i-1, v], \min_{w \in V}\{ M[i-1, w] + len(vw) \}\}$$

Shortest Path: Soundness and Running Time

Theorem

The ShortestPath algorithm correctly computes the minimum cost of an s-t path in any graph that has no negative cycles, and runs in O(n^3) time

Proof.

Soundness follows by induction from the recurrent relation for the optimal value.

DIY.

Running time:

We fill up a table with n^2 entries. Each of them requires O(n) time

Shortest Path: Soundness and Running Time

Theorem

The ShortestPath algorithm can be implemented in O(mn) time

A big improvement for sparse graphs

Proof.

Consider the computation of the array entry M[i,v]:

$$M[i,v] = \min\{ M[i-1, v], \min_{w \in V}\{ M[i-1, w] + len(vw) \}\}$$

We need only compute the minimum over all nodes w for which v has an edge to w

Let n_v denote the number of such edges

Shortest Path: Running Time Improvements

It takes O(n_v) to compute the array entry M[i,v].

It needs to be computed for every node v and for each i, 1 ≤ i ≤ n.

Thus the bound for running time is

$$O\left(n \sum_{v \in V} n_v \right) = O(nm)$$

Indeed, n_v is the outdegree of v, and we have the result by the Handshaking Lemma.

QED

Shortest Path: Space Improvements

The straightforward implementation requires storing a table with entries

It can be reduced to O(n)

Instead of recording M[i,v] for each i, we use and update a single value M[v] for each node v, the length of the shortest path from v to t found so far

Thus we use the following recurrent relation:

$M[v] = \min\{ M[v], \min_{w \in V} \{ M[w] + len(vw) \}\}$

Shortest Path: Space Improvements (cntd)

Lemma

Throughout the algorithm $M[v]$ is the length of some path from v to t, and after i rounds of updates the value $M[v]$ is no larger than the length of the shortest from v to t using at most i edges

Shortest Path: Finding Shortest Path

In the standard version we only need to keep record on how the optimum is achieved

Consider the space saving version.

For each node v store the first node on its path to the destination t

Denote it by first(v)

Update it every time $M[v]$ is updated

Let P be the pointer graph $P = (V, \{(v, \text{first}(v)) : v \in V\})$

Shortest Path: Finding Shortest Path

Lemma

If the pointer graph P contains a cycle C, then this cycle must have negative cost.

Proof

If w = first(v) at any time, then M[v] ≥ M[w] + len(vw)

Let $v_1, v_2, \ldots, v_k$ be the nodes along the cycle C, and (v_k, v_1) the last arc to be added

Consider the values right before this arc is added

We have $M[v_i] \geq M[v_{i+1}] + len(v_i v_{i+1})$ for i = 1,…, k – 1 and

$M[v_k] > M[v_1] + len(v_k v_1)$

Adding up all the inequalities we get $$0 > \sum_{i=1}^{k-1} len(v_i v_{i+1}) + len(v_k v_1)$$

Shortest Path: Finding Shortest Path (cntd)

Lemma

Suppose G has no negative cycles, and let P be the pointer graph after termination of the algorithm. For each node v, the path in P from v to t is a shortest v-t path in G.

Proof

Observe that P is a tree.

Since the algorithm terminates we have M[v] = M[w] + len(vw), where w = first(v).

As M[t] = 0, the length of the path traced out by the pointer graph is exactly M[v], which is the shortest path distance.

QED

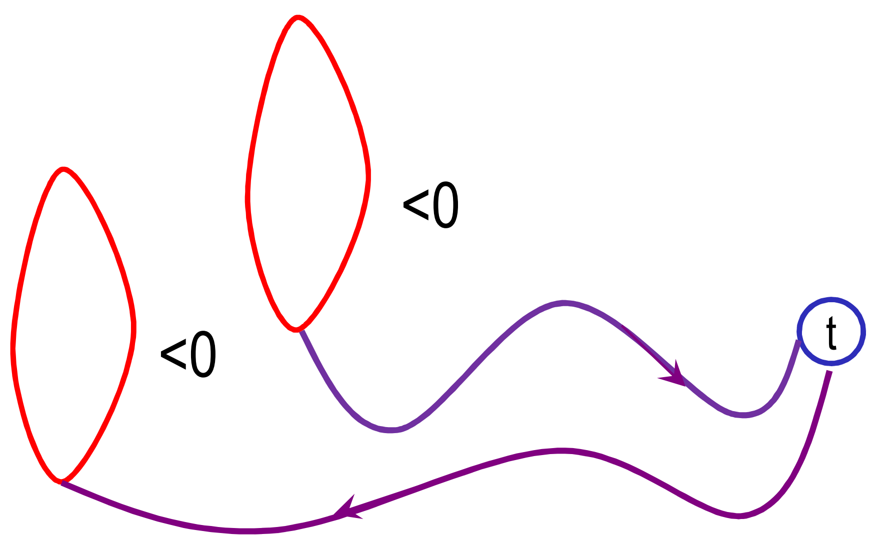
Shortest Path: Finding Negative Cycles

Two questions:

- how to decide if there is a negative cycle?
- how to find one?

Lemma

It suffices to find negative cycles C such that t can be reached from C

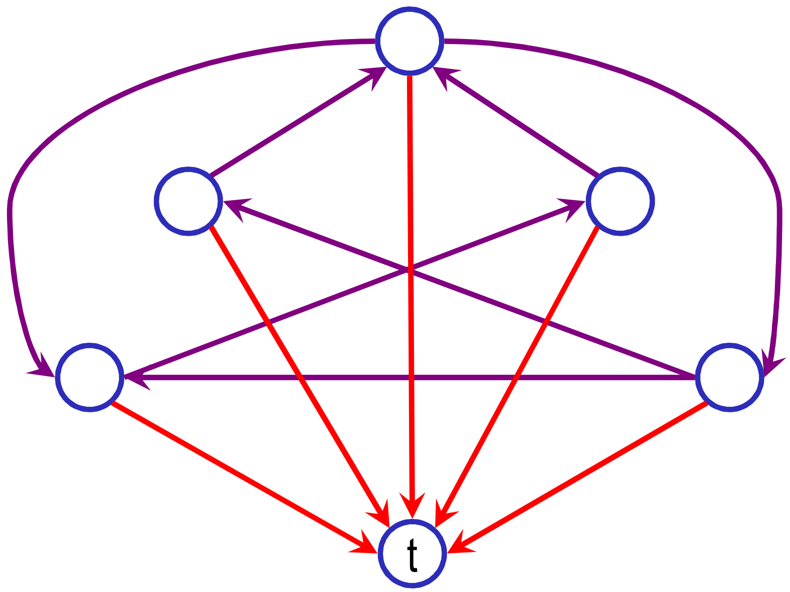


Shortest Path: Finding Negative Cycles

Proof

Let G be a graph

The augmented graph, A(G), is obtained by adding a new node and connecting every node in G with the new node



As is easily seen, G contains a negative cycle if and only if A(G) contains a negative cycle C such that t is reachable from C

QED

Shortest Path: Finding Negative Cycles (cntd)

Extend OPT(i,v) to $i \geq n$

If the graph G does not contain negative cycles then
OPT(i,v) = OPT(n – 1,v) for all nodes v and all $i \geq n$

Indeed, it follows from the observation that every shortest path contains at most n – 1 arcs.

Lemma

There is no negative cycle with a path to t if and only if
OPT(n,v) = OPT(n – 1,v)

Proof

If there is no negative cycle, then OPT(n,v) = OPT(n – 1,v) for all nodes v by the observation above

Shortest Path: Finding Negative Cycles (cntd)

Proof (cntd)

Suppose OPT(n,v) = OPT(n – 1,v) for all nodes v.

Therefore

$$\begin{aligned} OPT(n,v) &= \min\{ OPT(n-1,v),\ \min_{w \in V}\{ OPT(n-1,w) + len(vw) \}\} \\ &= \min\{ OPT(n,v),\ \min_{w \in V}\{ OPT(n,w) + len(vw) \}\} \\ &= OPT(n+1,v) \\ &= \ldots \end{aligned}$$

However, if a negative cycle from which t is reachable exists, then

$$\lim_{i \to \infty} OPT(i,v) = -\infty$$

Shortest Path: Finding Negative Cycles (cntd)

Let v be a node such that $OPT(n,v) \neq OPT(n-1,v)$.

A path P from v to t of weight $OPT(n,v)$ must use exactly n arcs

Any simple path can have at most $n-1$ arcs, therefore P contains a cycle C

Lemma

If G has n nodes and $OPT(n,v) \neq OPT(n-1,v)$, then a path P of weight $OPT(n,v)$ contains a cycle C, and C is negative.

Proof

Every path from v to t using less than n arcs has greater weight.

Let w be a node that occurs in P more than once.

Let C be the cycle between the two occurrences of w

Deleting C we get a shorter path of greater weight, thus C is negative