## Objectives

Data structures: Graphs

- DAGs and Topological order

Greedy Algorithms



## Strong Connectivity: Algorithm

Theorem. Can determine if G is strongly connected in O(m + n) time.

Pf. $\boxed{\text{Either DFS or BFS}}$

- Pick any node *s*
- Run BFS from *s* in G
- Run BFS from *s* in $G^{rev}$

reverse orientation of every edge in G
Or, the BFS using the *in* edges

- Return true iff all nodes reached in both BFS executions
- Correctness follows immediately from previous lemma
  - All reachable from one node, s is reached by all



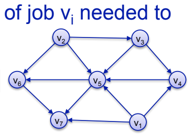
## Directed Acyclic Graphs

Def. A DAG is a directed graph that contains *no directed cycles*.

Example. Precedence constraints: edge $(v_i, v_j)$ means $v_i$ must precede $v_j$

- Course prerequisite graph: course $v_i$ must be taken before $v_j$
- Compilation: module $v_i$ must be compiled before $v_j$
- Pipeline of computing jobs: output of job $v_i$ needed to determine input of job $v_j$

a DAG:

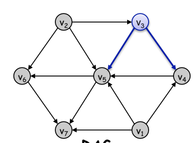




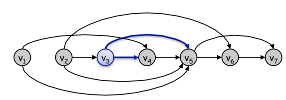
## Directed Acyclic Graphs

Given a set of tasks with dependencies, what is a valid order in which the tasks could be performed?

Def. A ***topological order*** of a directed graph G = (V, E) is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge $(v_i, v_j)$ we have $i < j$.



a DAG



**a topological ordering**
All edges point "forward"



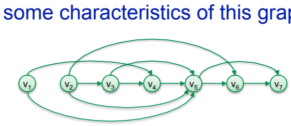
## Directed Acyclic Graphs

Does every DAG have a topological ordering?

- If so, how do we compute one?

What would we need to be able to create a topological ordering?

- What are some characteristics of this graph?



Need some place to start … Where?



## Directed Acyclic Graphs

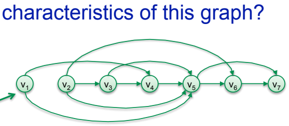
Does every DAG have a topological ordering?

- If so, how do we compute one?

What would we need to be able to create a topological ordering?

- What are some characteristics of this graph?

Need someplace to start: a node with no incoming edges (no dependencies)



Note that both $v_1$ and $v_2$ have no incoming edges

## Directed Acyclic Graphs

Does every DAG have a node with no incoming edges?



## Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges

- That node is our starting point of the topological ordering

How to prove?



## Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges

Proof idea: consider if there is no node without incoming edges

- What does that mean?
- Recall that we know that G is a DAG
  - What are its properties?

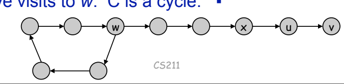


## Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge
- Pick any node *v*, and follow edges backward from *v*
  - Since *v* has at least one incoming edge (u, v), we can walk backward to *u*
- Since *u* has at least one incoming edge (x, u), we can walk backward to *x*
- Repeat until we visit a node, say *w*, twice
  - Has to happen at least by *n+1* steps (What if can't go n+1 steps?)
- Let C denote the sequence of nodes encountered between successive visits to *w*. C is a cycle. ▪






## Creating a Topological Order

With a node with no incoming edges, can create a topological ordering

Think about a DAG with only one node. What is its topological ordering?

Only two nodes?

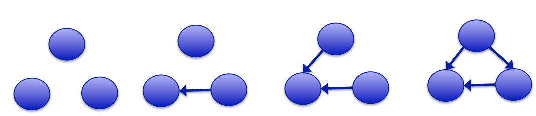
Three nodes?

- What are the DAG, TO possibilities?



## Topological Order for Three Nodes

What are the possibilities?



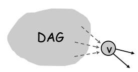
Can't add any more edges without creating a cycle.

## Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

Pf. (by induction on n)

- Base case: true if n = 1
- Given DAG on n > 1 nodes, find a node *v* with no incoming edges
- G - { *v* } is a DAG, since deleting *v* cannot create cycles
- By inductive hypothesis, G - { *v* } has a topological ordering
- Place *v* first in topological ordering; then append nodes of G - { *v* }
- in topological order. This is valid since *v* has no incoming edges. ▪



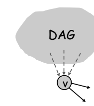



## Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

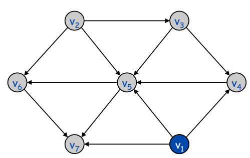
Algorithm:

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```





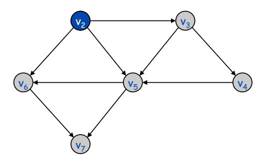

## Topological Ordering Algorithm: Example



Topological order:



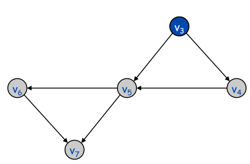
## Topological Ordering Algorithm: Example



Topological order: $v_1$



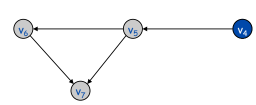
## Topological Ordering Algorithm: Example



Topological order: $v_1$, $v_2$

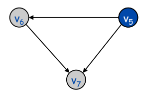


## Topological Ordering Algorithm: Example



Topological order: $v_1$, $v_2$, $v_3$

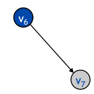
## Topological Ordering Algorithm: Example



Topological order: $v_1, v_2, v_3, v_4$



## Topological Ordering Algorithm: Example



Topological order: $v_1, v_2, v_3, v_4, v_5$

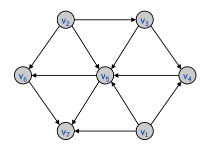


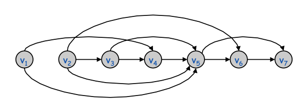
## Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$



## Topological Ordering Algorithm: Example





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.



## Topological Order Runtime

Where are the costs?

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```



## Topological Order Runtime

Where are the costs?

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

Find a node without incoming edges and delete it:
  O(n)

Repeat on all nodes

→ $O(n^2)$

Can we do better?

## Topological Sorting Algorithm: Running Time

Theorem. Find a topological order in O(m + n) time

Pf.

- Maintain the following information:
  - `count[w]` = remaining number of incoming edges
  - S = set of remaining nodes with no incoming edges
- Initialization: O(m + n) via single scan through graph
- Update: to delete v
  - remove v from S
  - decrement `count[w]` for all edges from v to w
    - add w to S if c `count[w]` hits 0
  - O(1) per edge ▪



## GREEDY ALGORITHMS



## Greedy Algorithms

At each step

- Take as much as you can get
  - "local" optimizations



## Example of Greedy Algorithm

How do you make change to give out the fewest coins?



## Example of Greedy Algorithm

How do you make change to give out the fewest coins?

- Local optimum: coin of the highest value, less than the remaining change owed

```
while change > 0:
    if change >= 25:
        print "Quarter"
        change -= 25
    elif change >= 10:
        print "Dime"
        change -= 10
    …
```



## Proving Greedy Algorithms Work

Specifically, produce an **optimal** solution

Two approaches:

- Greedy algorithm stays ahead
  - Does better than any other algorithm at each step
- Exchange argument
  - Transform any solution into a greedy solution

Greedy algorithm stays ahead

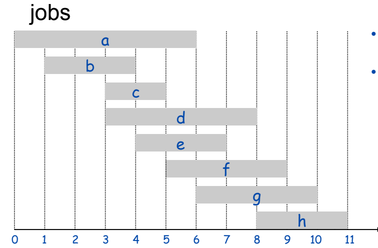
# INTERVAL SCHEDULING

## Interval Scheduling

Job j starts at $s_j$ and finishes at $f_j$

Two jobs *compatible* if they don't overlap

**Goal**: find maximum subset of mutually compatible jobs




- Every job is worth equal money.
- To earn the most money → schedule the most jobs

32

## Greedy Algorithm Template

Consider jobs (or whatever) in some order

- Decision: what order is best

Take each job provided it's compatible with the ones already taken

What are options for orders?

What is our goal? What are we trying to minimize/maximize?

What is the worst case?

Feb 4, 2009 CS211 33

## Interval Scheduling: Greedy Algorithms

Earliest start time. Consider jobs in ascending order of start time $s_j$

- Utilize CPU as soon as possible

Earliest finish time. Consider jobs in ascending order of finish time $f_j$

- Resource becomes free ASAP
- Maximize time left for other requests

Shortest interval. Consider jobs in ascending order of interval length $f_j - s_j$

Fewest conflicts. For each job, count number of conflicting jobs $c_j$. Schedule in ascending order of conflicts $c_j$

34

## Interval Scheduling: Greedy Algorithms

Not optimal when …

breaks earliest start time

breaks shortest interval

breaks fewest conflicts

35

## Interval Scheduling: Greedy Algorithm

Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

jobs selected

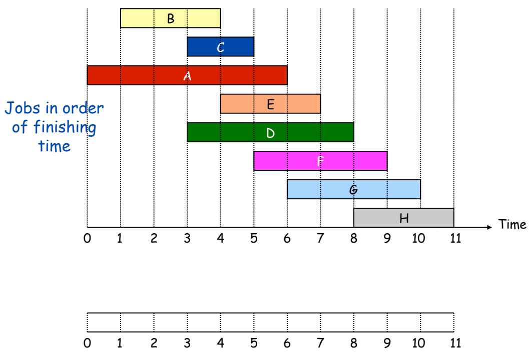
```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn

A = {}
for j = 1 to n
   if (job j compatible with A)
      A = A ∪ {j}
return A
```

Runtime of algorithm?

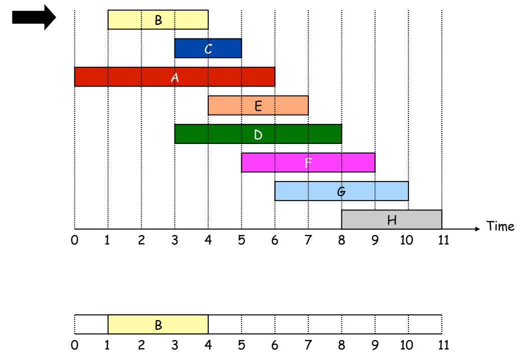
- Where/what are the costs?

36

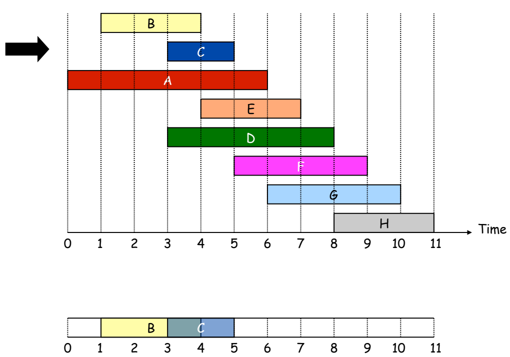
## Interval Scheduling



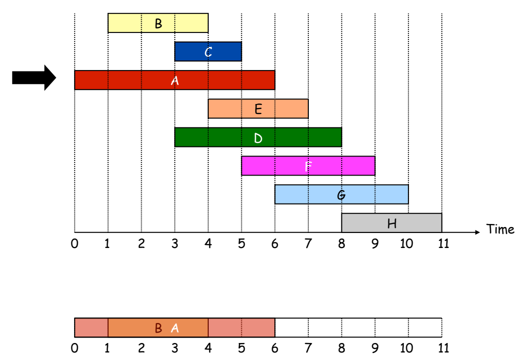

## Interval Scheduling



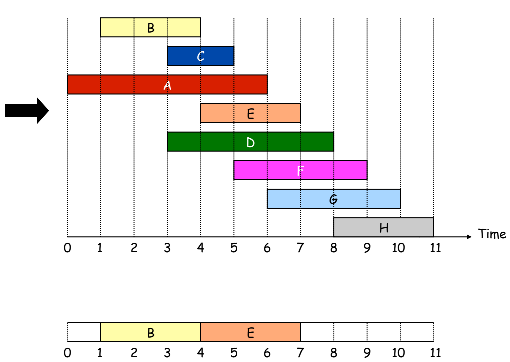

## Interval Scheduling



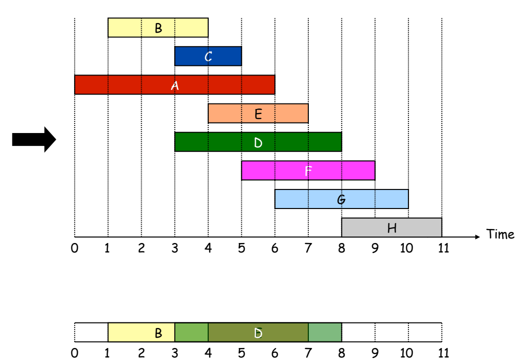

## Interval Scheduling



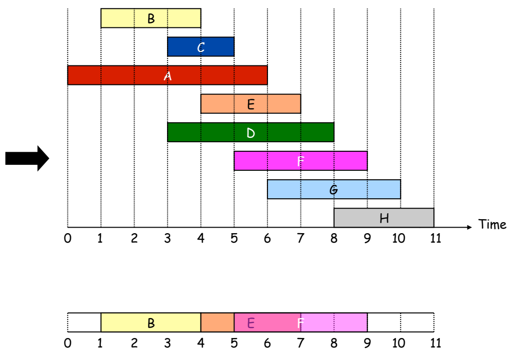

## Interval Scheduling



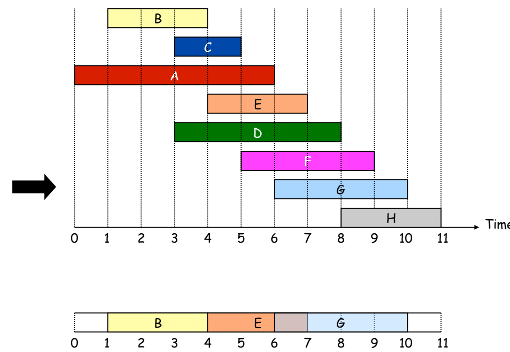

## Interval Scheduling



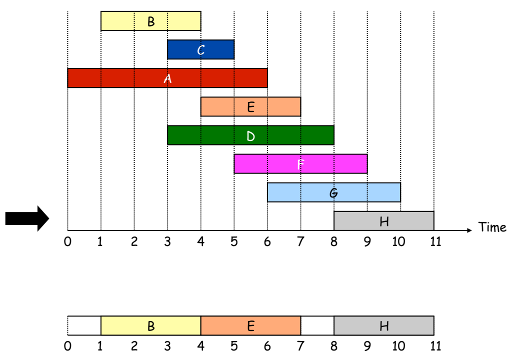
## Interval Scheduling




## Interval Scheduling




## Interval Scheduling




## Interval Scheduling: Greedy Algorithm

Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

jobs selected

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

A = {}
for j = 1 to n
   if (job j compatible with A)
      A = A ∪ {j}
return A
```

Implementation. O(n log n)

- Remember job j* that was added last to A
- Job j is compatible with A if $s_j \geq f_{j^*}$.

46

## Interval Scheduling: Analysis

Know that the intervals are compatible

- Handle by the if statement

But is it optimal?

- What are we looking for?

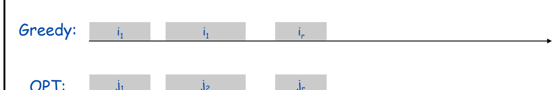
## Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy (*k* jobs)
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution (*m* jobs)
- Same ordering, by finish times

➡Want to show that $k = m$




What can we say about $i_1$ and $j_1$? $f(i_1) <= f(j_1)$

48