CSEP 521
Applied Algorithms

Richard Anderson

Winter 2013

Lecture 2

Announcements

- Reading
  - Chapter 2.1, 2.2
  - Chapter 3
  - Chapter 4
- Homework Guidelines
  - Prove that your algorithm works
    - A proof is a “convincing argument”
  - Give the run time for you algorithm
    - Justify that the algorithm satisfies the runtime bound
  - You may lose points for style

Announcements

- Monday, January 21 is a holiday
  - No class
- Makeup lecture, Thursday, January 17, 5:00 pm – 6:30 pm
  - UW and Microsoft
  - View off line if you cannot attend
- Homework 2 is due January 21
  - Electronic turn in only

What does it mean for an algorithm to be efficient?

Definitions of efficiency

- Fast in practice
- Qualitatively better worst case performance than a brute force algorithm

Polynomial time efficiency

- An algorithm is efficient if it has a polynomial run time
- Run time as a function of problem size
  - Run time: count number of instructions executed on an underlying model of computation
  - T(n): maximum run time for all problems of size at most n

Polynomial Time

- Algorithms with polynomial run time have the property that increasing the problem size by a constant factor increases the run time by at most a constant factor (depending on the algorithm)

Why Polynomial Time?

- Generally, polynomial time seems to capture the algorithms which are efficient in practice

- The class of polynomial time algorithms has many good, mathematical properties

Polynomial vs. Exponential Complexity

- Suppose you have an algorithm which takes n! steps on a problem of size n
- If the algorithm takes one second for a problem of size 10, estimate the run time for the following problems sizes:

12 14 16 18 20

Ignoring constant factors

- Express run time as O(f(n))
- Emphasize algorithms with slower growth rates
- Fundamental idea in the study of algorithms
- Basis of Tarjan/Hopcroft Turing Award

Why ignore constant factors?

- Constant factors are arbitrary
  - Depend on the implementation
  - Depend on the details of the model

- Determining the constant factors is tedious and provides little insight

Why emphasize growth rates?

- The algorithm with the lower growth rate will be faster for all but a finite number of cases
- Performance is most important for larger problem size
- As memory prices continue to fall, bigger problem sizes become feasible
- Improving growth rate often requires new techniques

Formalizing growth rates

- T(n) is O(f(n)) [T : $Z^+ \rightarrow R^+$]
  - If n is sufficiently large, T(n) is bounded by a constant multiple of f(n)
  - Exist c, n_0, such that for n > n_0, T(n) < c f(n)

- T(n) is O(f(n)) will be written as:
  T(n) = O(f(n))
  - Be careful with this notation

Prove $3n^2 + 5n + 20$ is $O(n^2)$

Let $c =$

Let $n_0 =$

$T(n)$ is $O(f(n))$ if there exist c, n_0, such that for $n > n_0$, $T(n) < c\, f(n)$

Order the following functions in increasing order by their growth rate

a) $n \log^4 n$

b) $2n^2 + 10n$

c) $2^{n/100}$

d) $1000n + \log^8 n$

e) n^{100}

f) 3^n

g) $1000 \log^{10} n$

h) $n^{1/2}$

Lower bounds

- T(n) is Ω(f(n))
  - T(n) is at least a constant multiple of f(n)
  - There exists an n_0, and $\varepsilon > 0$ such that T(n) > εf(n) for all $n > n_0$
- Warning: definitions of Ω vary

- T(n) is Θ(f(n)) if T(n) is O(f(n)) and T(n) is Ω(f(n))

Useful Theorems

- If lim (f(n) / g(n)) = c for c > 0 then f(n) = Θ(g(n))

- If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) is O(h(n))

- If f(n) is O(h(n)) and g(n) is O(h(n)) then f(n) + g(n) is O(h(n))

Ordering growth rates

- For b > 1 and x > 0
  - $\log^b n$ is $O(n^x)$
- For r > 1 and d > 0
  - n^d is $O(r^n)$

Stable Matching

Reported Results

Student	n	M / n	W / n	M / n * W / n
Stanislav	10,000	9.96	1020	10159
Andy	4,096	8.77	472	4139
Boris	5,000	10.06	499	5020
Huy	10,000	10.68	969	10349
Hans	10,000	9.59	1046	10031
Vijayanand	1,000	8.60	114	980
Robert	20,000	12.40	1698	21055
Zain	2,825	8.61	331	2850
Uzair	8,192	9.10	883	8035
Anand	10,000	9.58	1045	10011

Why is $M/n \sim \log n$?

Why is W/n ~ n / log n?

Graph Theory

Graph Theory

- G = (V, E)
  - V – vertices
  - E – edges
- Undirected graphs
  - Edges sets of two vertices {u, v}
- Directed graphs
  - Edges ordered pairs (u, v)
- Many other flavors
  - Edge / vertices weights
  - Parallel edges
  - Self loops

Definitions

- Path: $v_1, v_2, \ldots, v_k$, with (v_i, v_{i+1}) in E
  - Simple Path
  - Cycle
  - Simple Cycle
- Distance
- Connectivity
  - Undirected
  - Directed (strong connectivity)
- Trees
  - Rooted
  - Unrooted

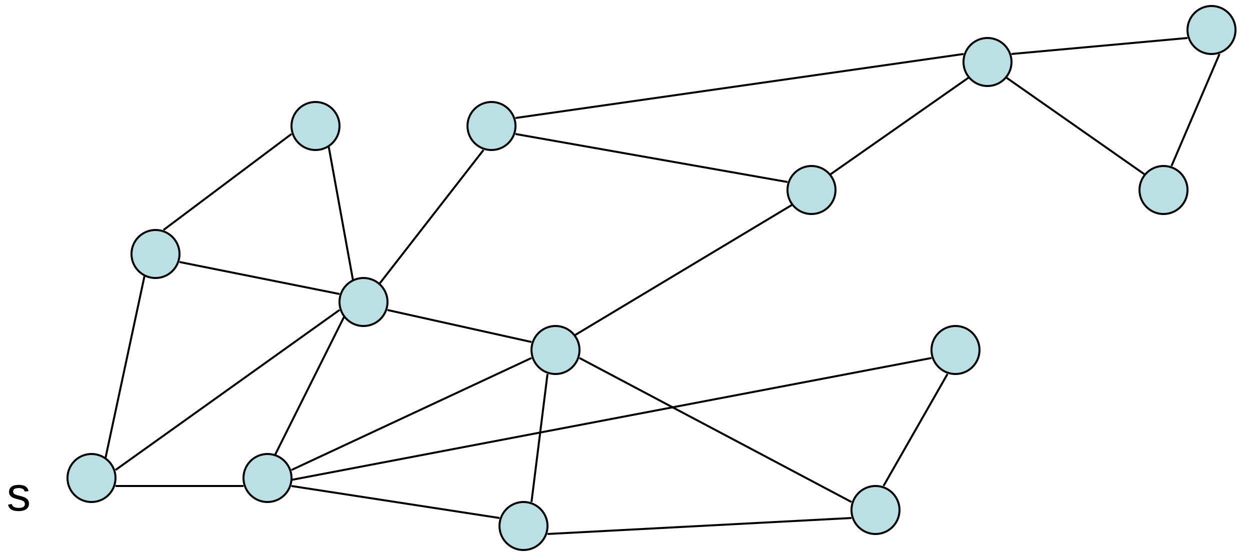
Graph search

- Find a path from s to t

```
S = {s}
While there exists (u, v) in E with u in S and v not in S
        Pred[v] = u
        Add v to S
        if (v = t) then path found
```

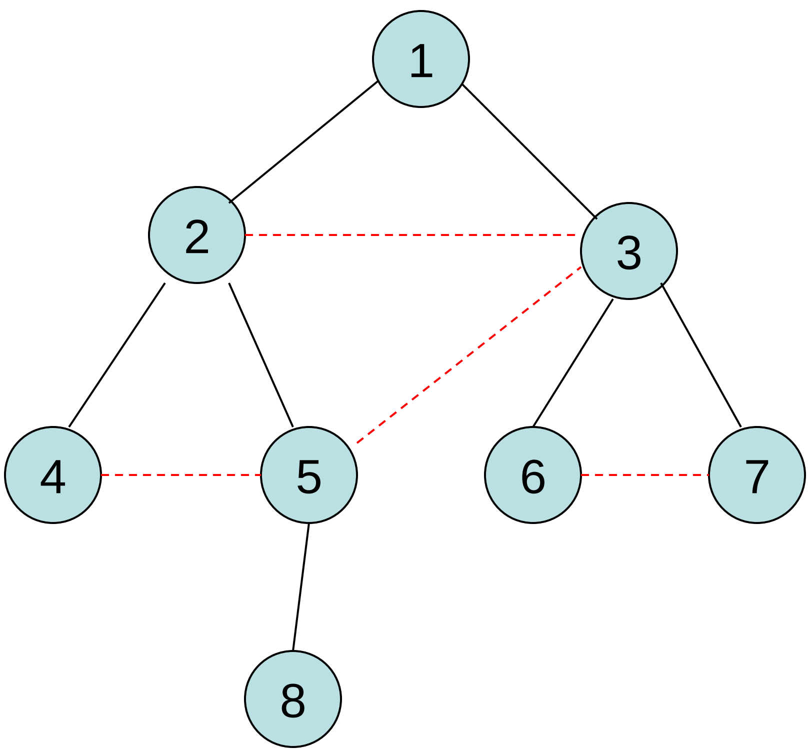
Breadth first search

- Explore vertices in layers
  - s in layer 1
  - Neighbors of s in layer 2
  - Neighbors of layer 2 in layer 3 . . .



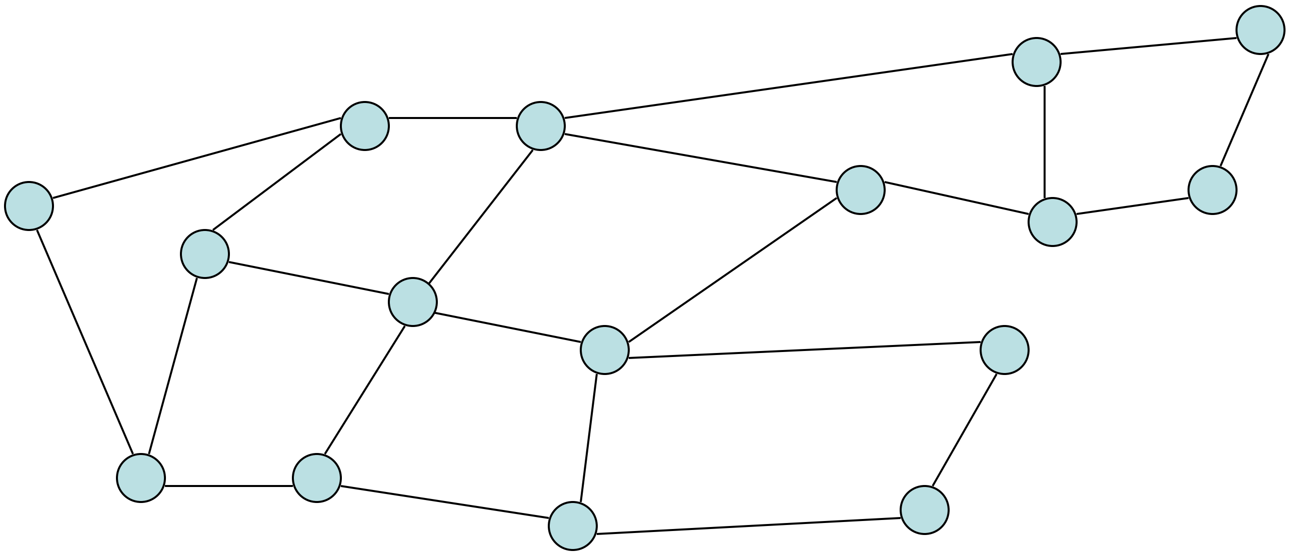
Key observation

- All edges go between vertices on the same layer or adjacent layers

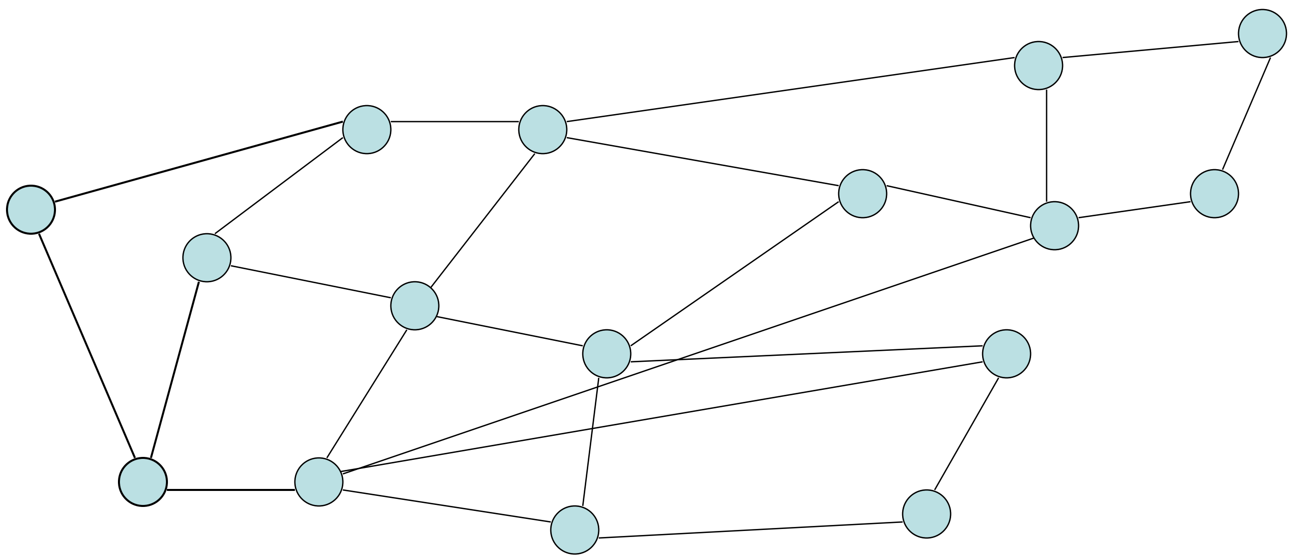


Bipartite Graphs

- A graph V is bipartite if V can be partitioned into V_1, V_2 such that all edges go between V_1 and V_2
- A graph is bipartite if it can be two colored



Can this graph be two colored?



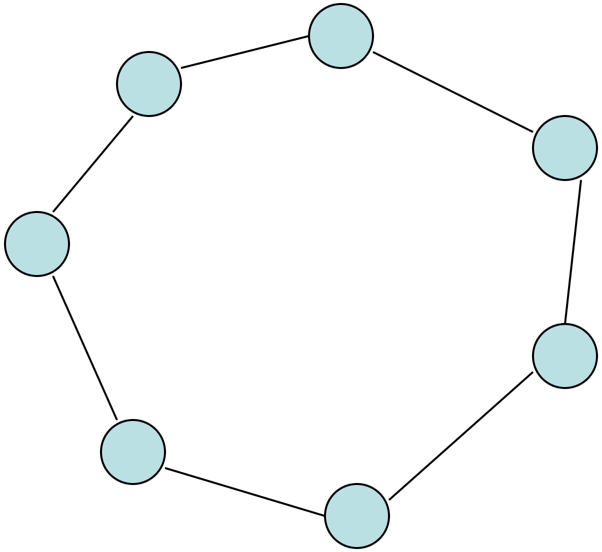
Algorithm

- Run BFS
- Color odd layers red, even layers blue
- If no edges between the same layer, the graph is bipartite
- If edge between two vertices of the same layer, then there is an odd cycle, and the graph is not bipartite

Theorem: A graph is bipartite if and only if it has no odd cycles

Lemma 1

- If a graph contains an odd cycle, it is not bipartite



Lemma 2

- If a BFS tree has an *intra-level edge*, then the graph has an odd length cycle

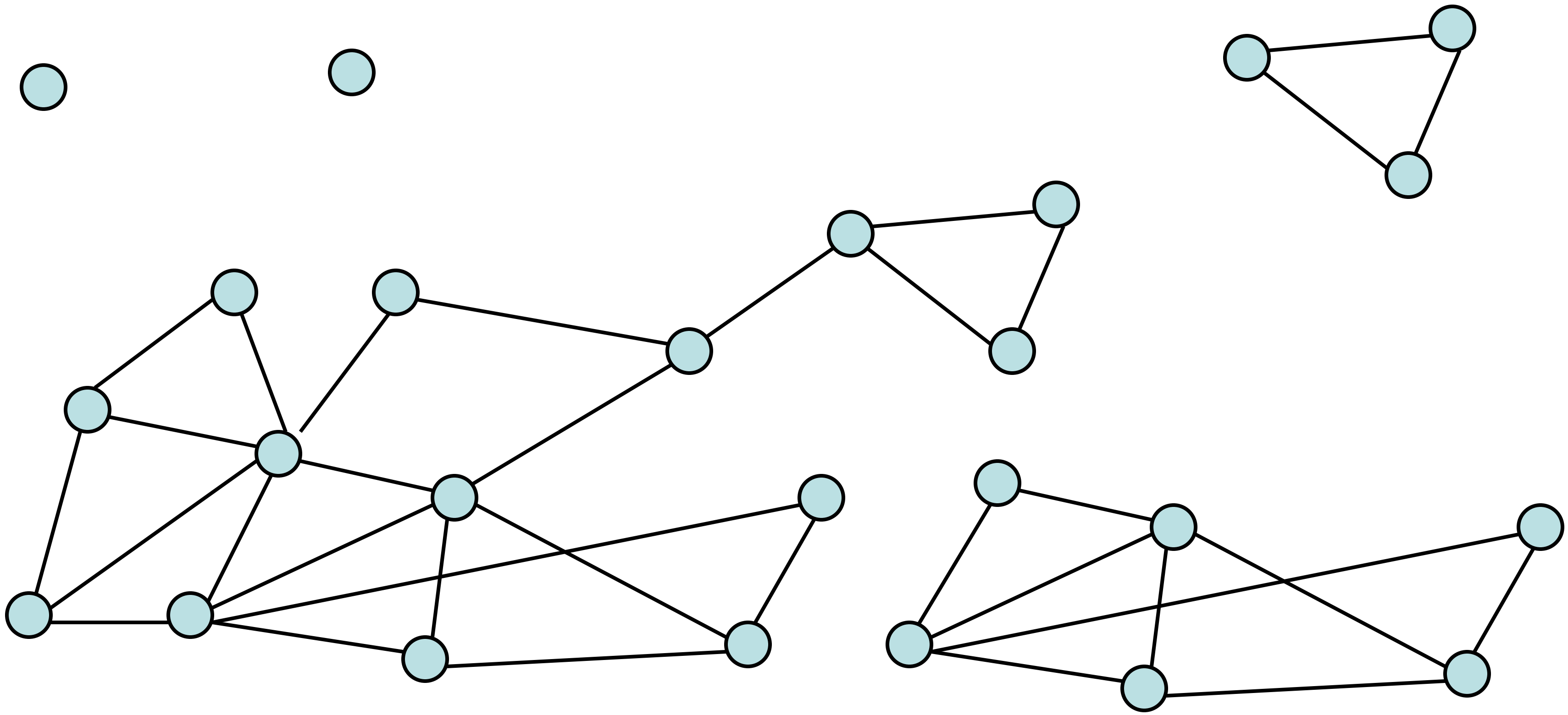
Intra-level edge: both end points are in the same level

Lemma 3

- If a graph has no odd length cycles, then it is bipartite

Connected Components

- Undirected Graphs

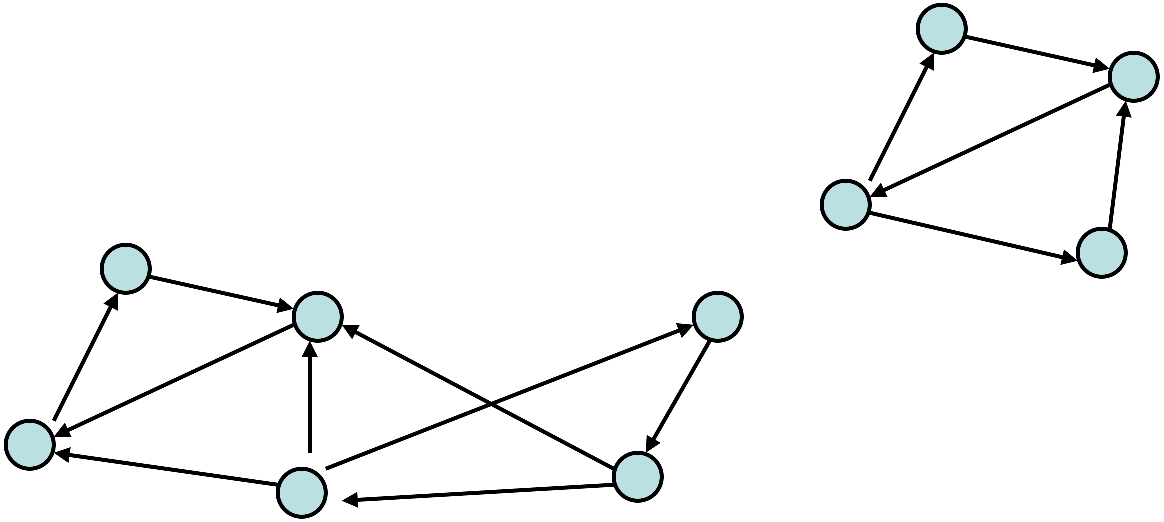


Computing Connected Components in O(n+m) time

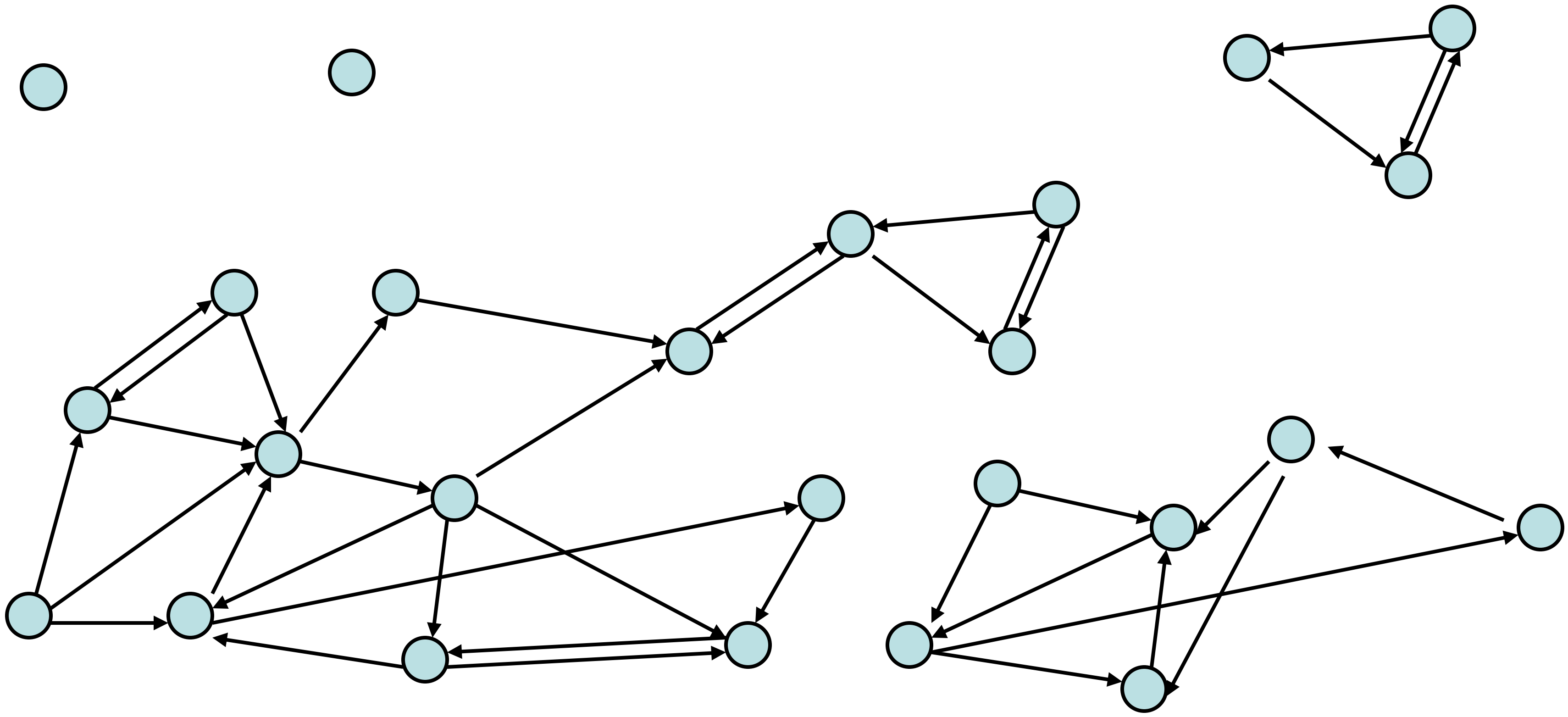
- A search algorithm from a vertex v can find all vertices in v's component
- While there is an unvisited vertex v, search from v to find a new component

Directed Graphs

- A Strongly Connected Component is a subset of the vertices with paths between every pair of vertices.



Identify the Strongly Connected Components

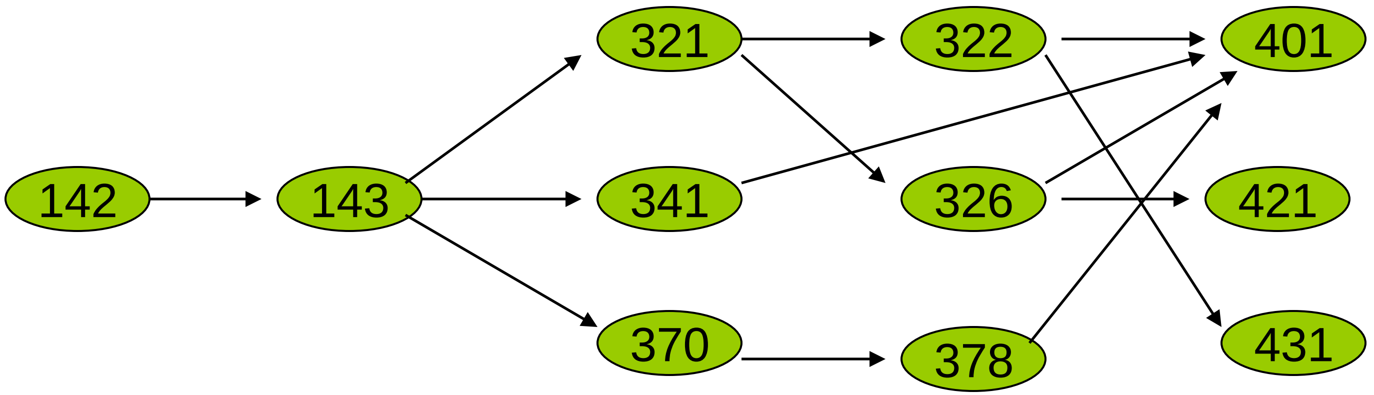


Strongly connected components can be found in O(n+m) time

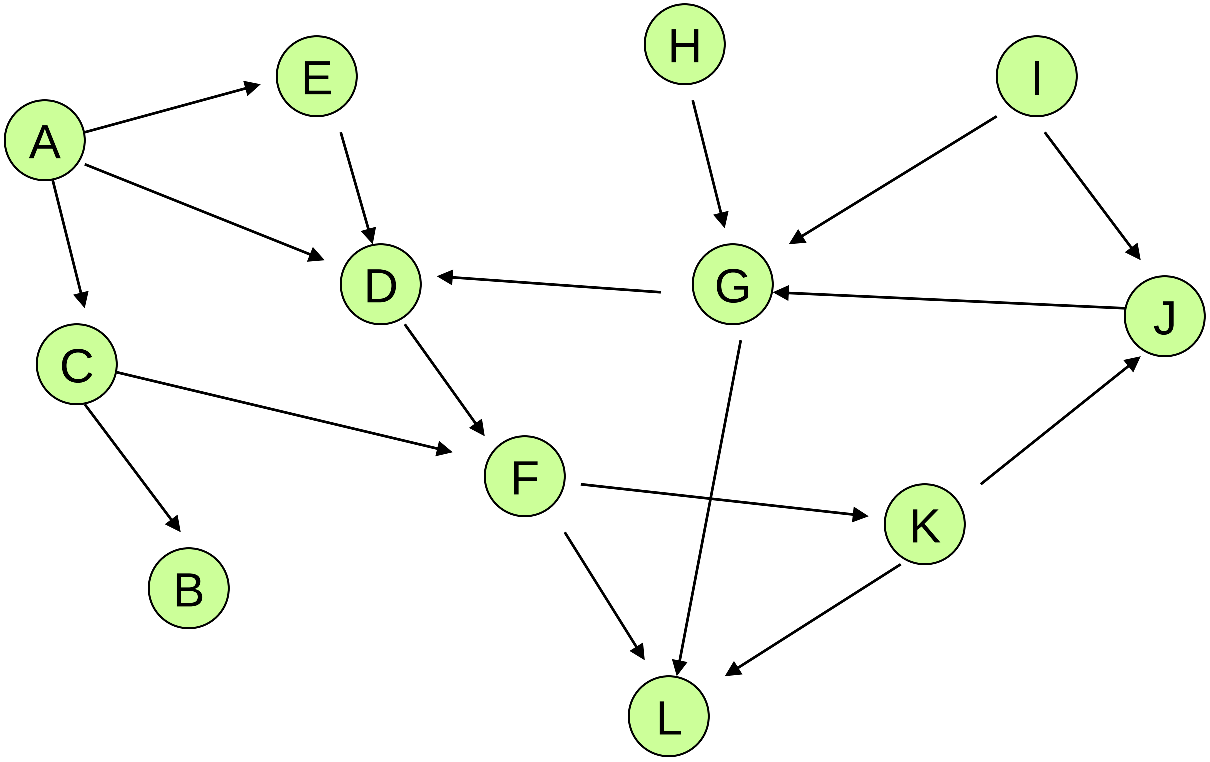
- But it's tricky!
- Simpler problem: given a vertex v, compute the vertices in v's scc in O(n+m) time

Topological Sort

- Given a set of tasks with precedence constraints, find a linear order of the tasks

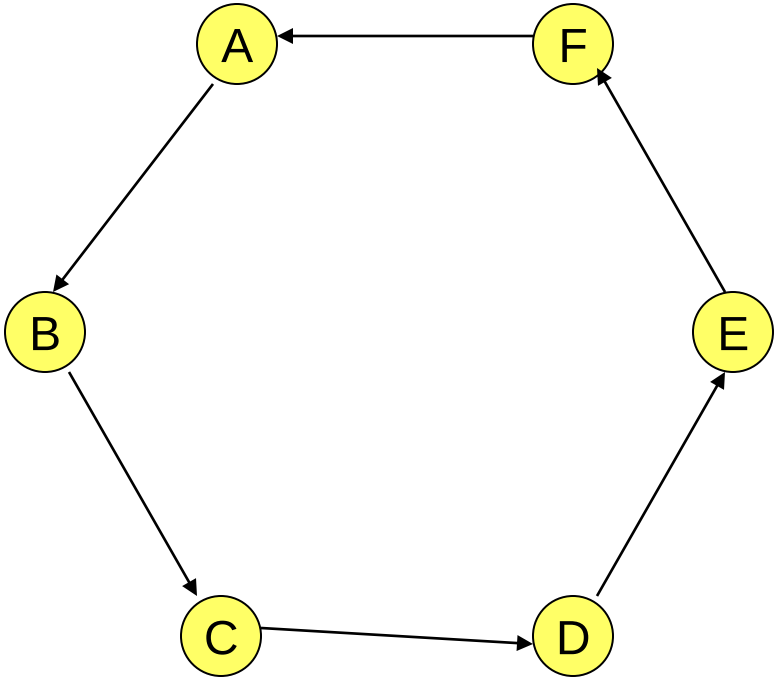


Find a topological order for the following graph



If a graph has a cycle, there is no topological sort

- Consider the first vertex on the cycle in the topological sort
- It must have an incoming edge



Lemma: If a graph is acyclic, it has a vertex with in degree 0

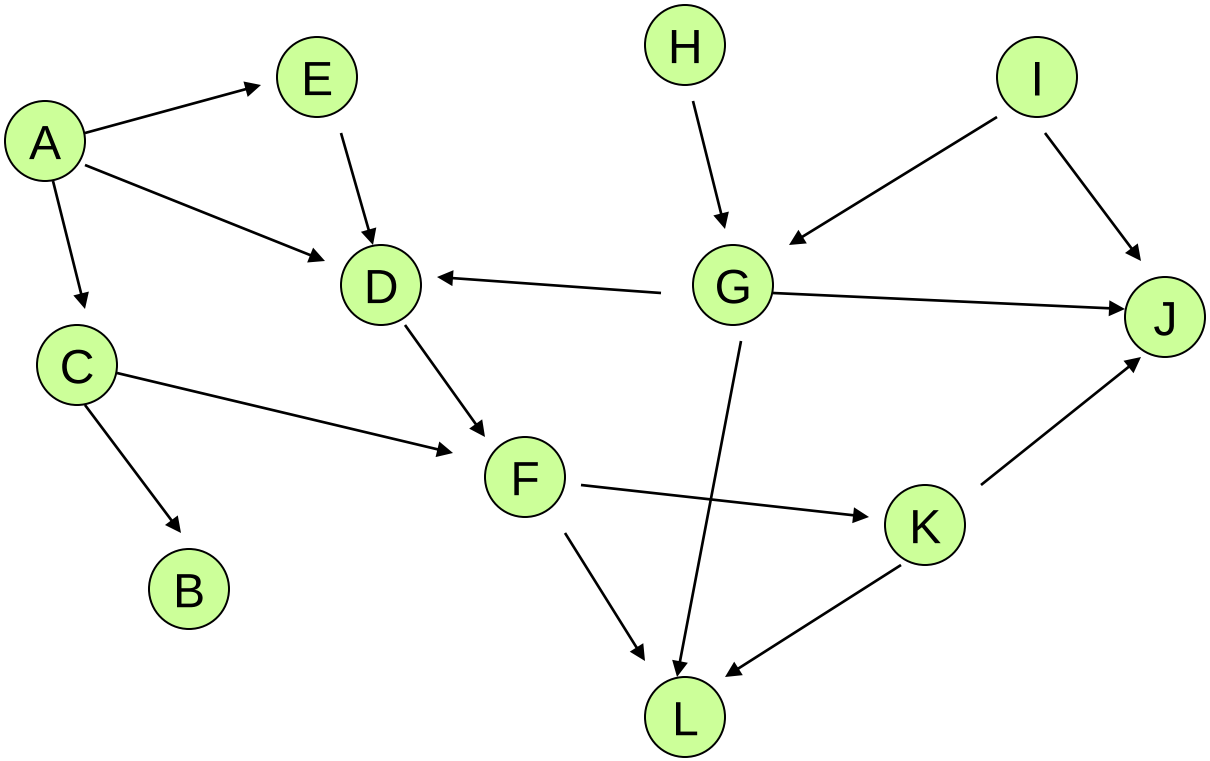
- Proof:
  - Pick a vertex v_1, if it has in-degree 0 then done
  - If not, let (v_2, v_1) be an edge, if v_2 has in-degree 0 then done
  - If not, let (v_3, v_2) be an edge . . .
  - If this process continues for more than n steps, we have a repeated vertex, so we have a cycle

Topological Sort Algorithm

While there exists a vertex v with in-degree 0

Output vertex v

Delete the vertex v and all out going edges

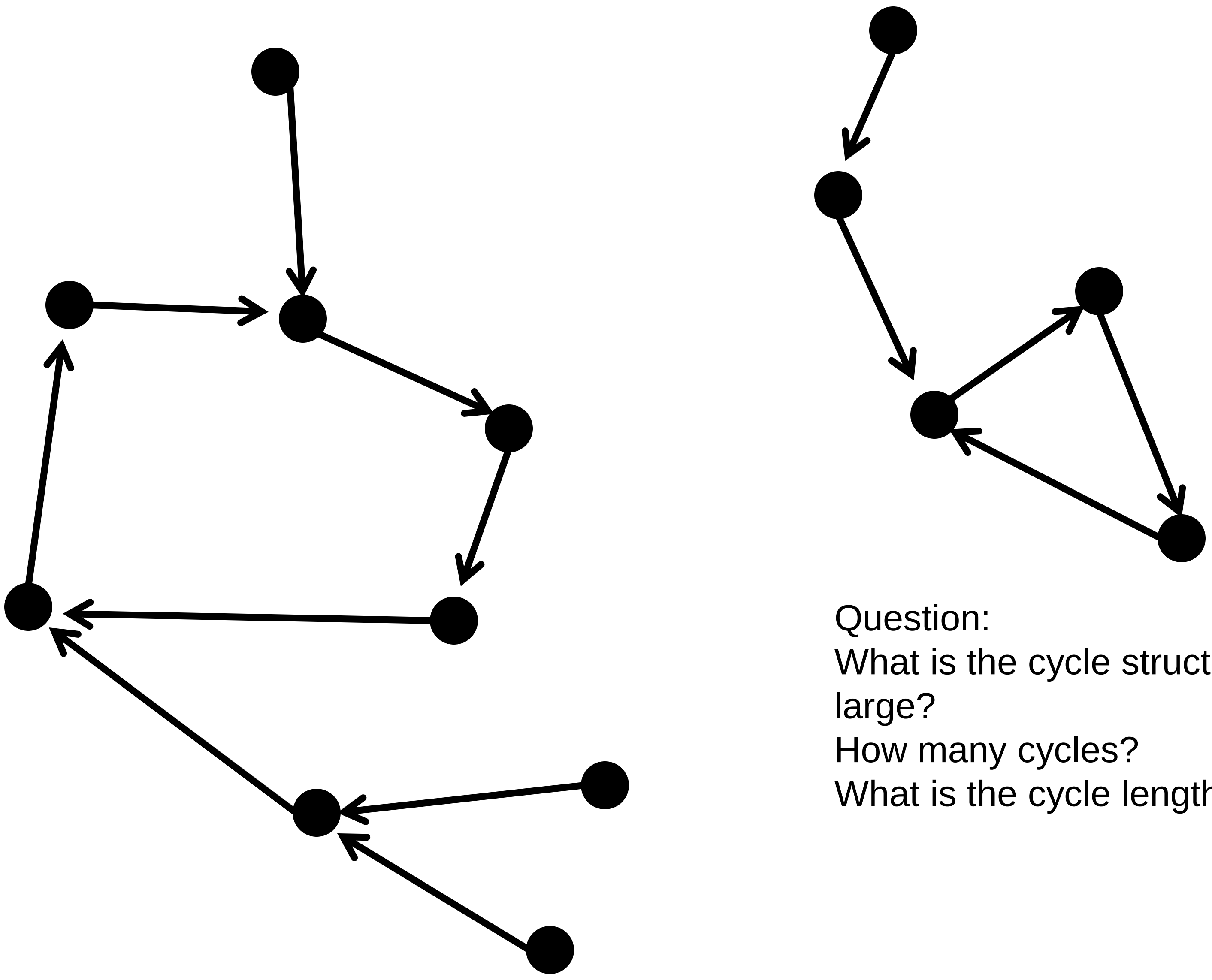


Details for O(n+m) implementation

- Maintain a list of vertices of in-degree 0
- Each vertex keeps track of its in-degree
- Update in-degrees and list when edges are removed
- m edge removals at O(1) cost each

Random Graph models

Random out degree one graph



Question:
What is the cycle structure as N gets large?
How many cycles?
What is the cycle length?

Greedy Algorithms

Greedy Algorithms

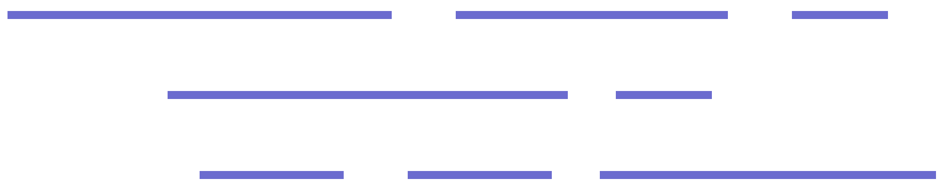
- Solve problems with the simplest possible algorithm
- The hard part: showing that something simple actually works
- Pseudo-definition
  - An algorithm is Greedy if it builds its solution by adding elements one at a time using a simple rule

Scheduling Theory

- Tasks
  - Processing requirements, release times, deadlines
- Processors
- Precedence constraints
- Objective function
  - Jobs scheduled, lateness, total execution time

Interval Scheduling

- Tasks occur at fixed times
- Single processor
- Maximize number of tasks completed



- Tasks {1, 2, . . . N}
- Start and finish times, s(i), f(i)

What is the largest solution?

Greedy Algorithm for Scheduling

Let T be the set of tasks, construct a set of independent tasks I, A is the rule determining the greedy algorithm

I = { }

While (T is not empty)

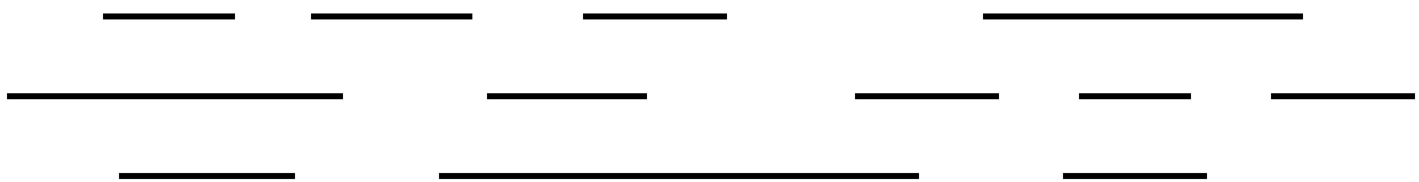
Select a task t from T by a rule A

Add t to I

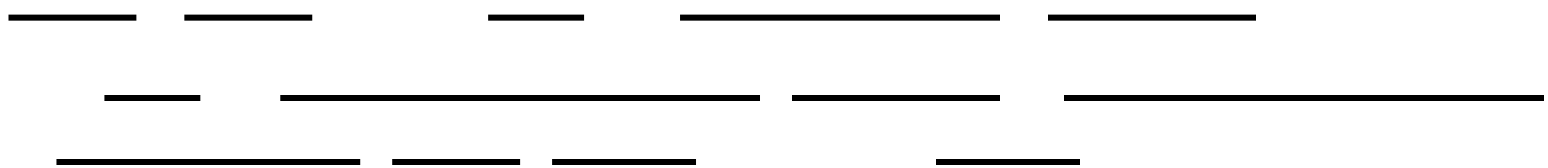
Remove t and all tasks incompatible with t from T

Simulate the greedy algorithm for each of these heuristics

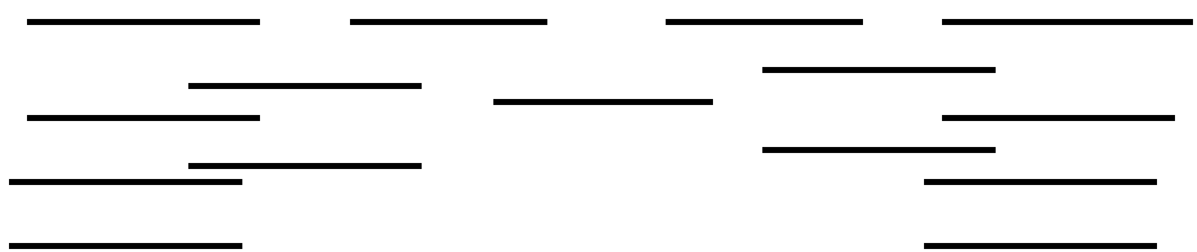
Schedule earliest starting task



Schedule shortest available task

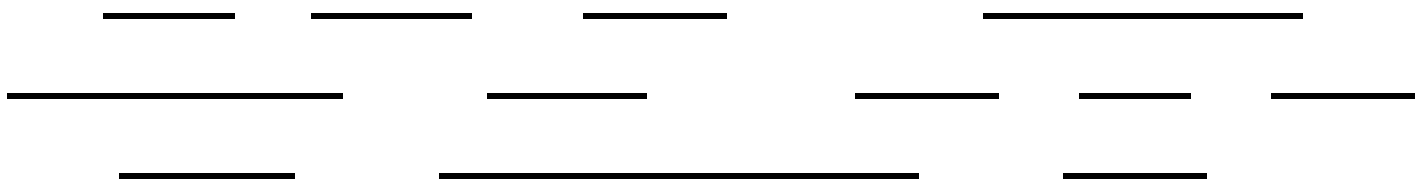


Schedule task with fewest conflicting tasks

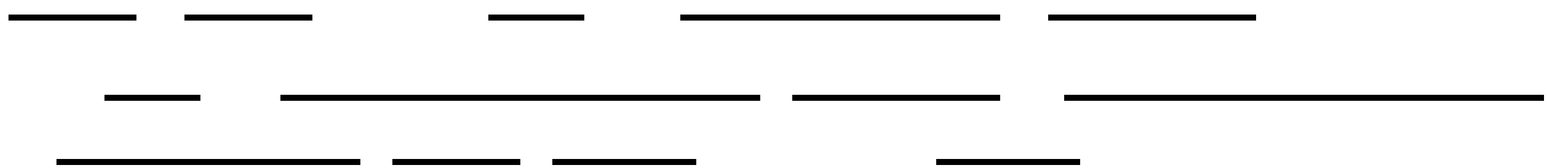


Greedy solution based on earliest finishing time

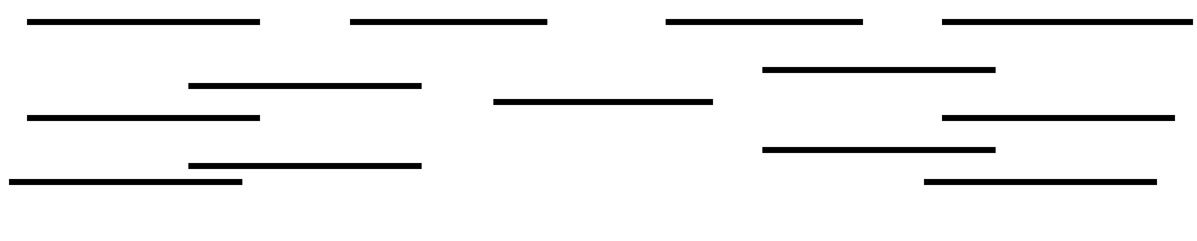
Example 1



Example 2



Example 3



Theorem: Earliest Finish Algorithm is Optimal

- Key idea: Earliest Finish Algorithm stays ahead
- Let $A = \{i_1, . . ., i_k\}$ be the set of tasks found by EFA in increasing order of finish times
- Let $B = \{j_1, . . ., j_m\}$ be the set of tasks found by a different algorithm in increasing order of finish times
- Show that for $r \le \min(k, m)$, $f(i_r) \le f(j_r)$

Stay ahead lemma

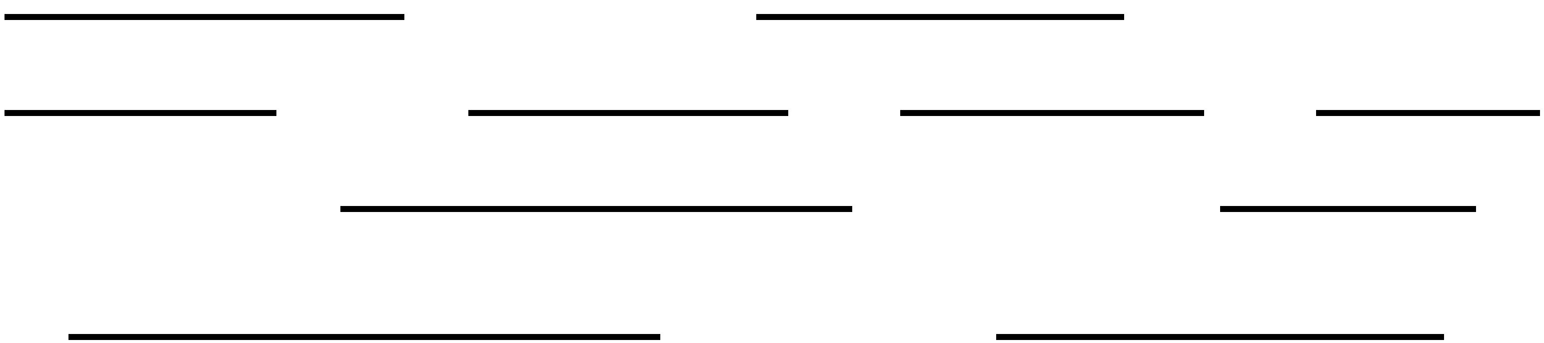
- A always stays ahead of B, $f(i_r) <= f(j_r)$
- Induction argument
  - $f(i_1) <= f(j_1)$
  - If $f(i_{r-1}) <= f(j_{r-1})$ then $f(i_r) <= f(j_r)$

Completing the proof

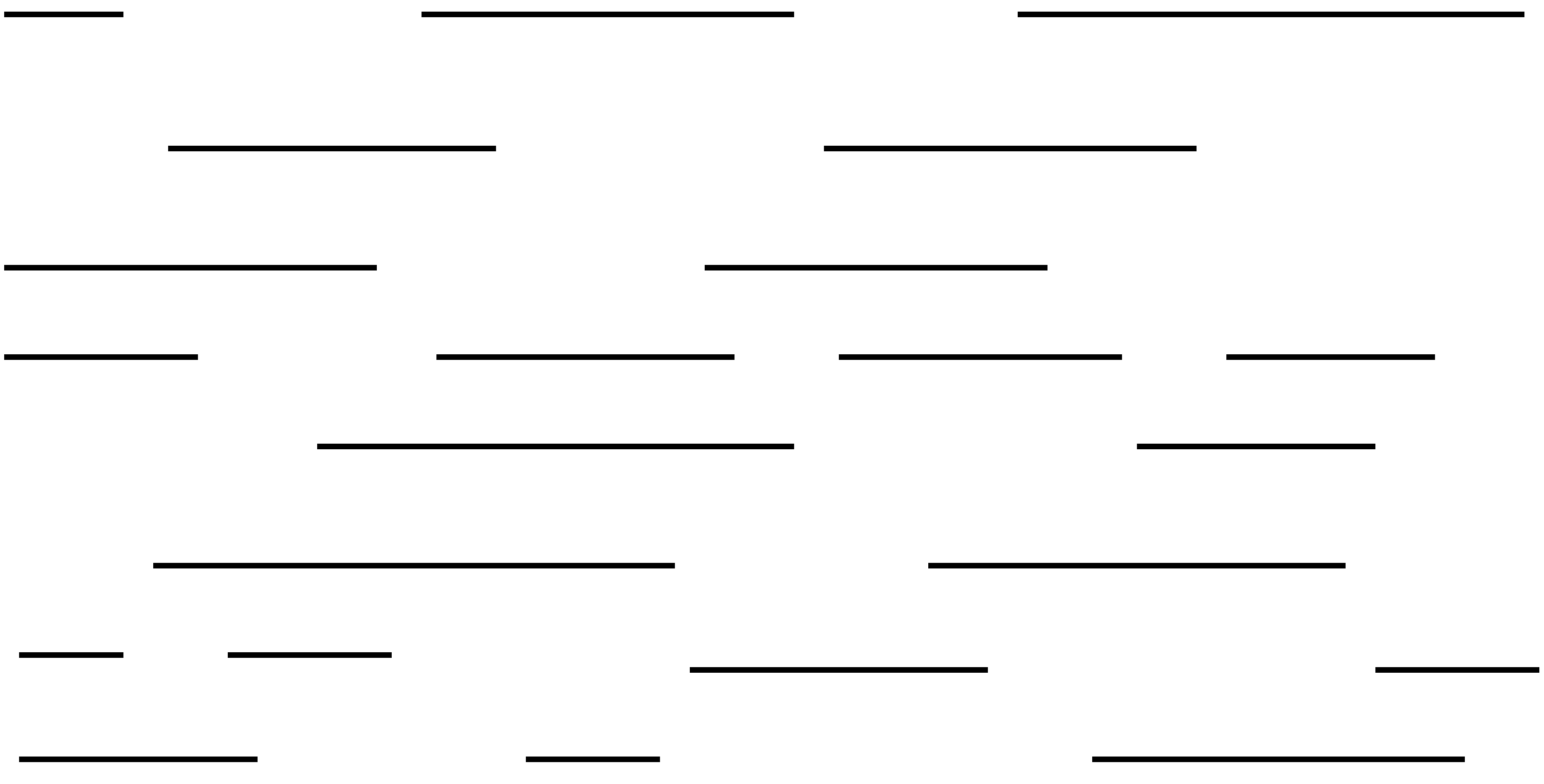
- Let A = $\{i_1, . . ., i_k\}$ be the set of tasks found by EFA in increasing order of finish times
- Let O = $\{j_1, . . ., j_m\}$ be the set of tasks found by an optimal algorithm in increasing order of finish times
- If $k < m$, then the Earliest Finish Algorithm stopped before it ran out of tasks

Scheduling all intervals

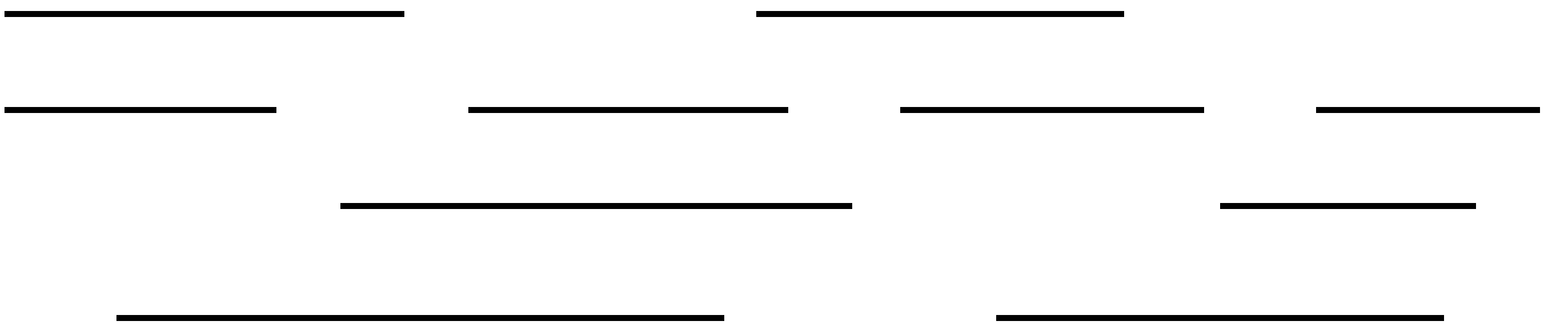
- Minimize number of processors to schedule all intervals



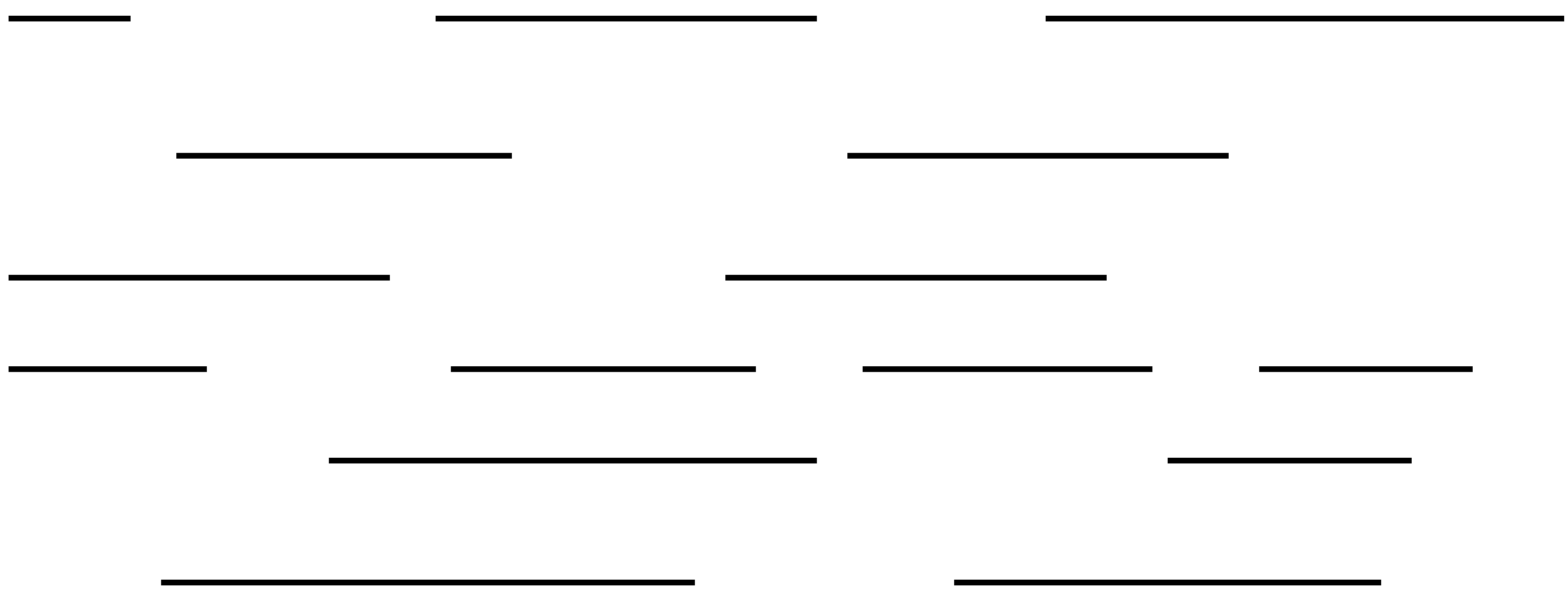
How many processors are needed for this example?



Prove that you cannot schedule this set of intervals with two processors



Depth: maximum number of intervals active



Algorithm

- Sort by start times
- Suppose maximum depth is d, create d slots
- Schedule items in increasing order, assign each item to an open slot

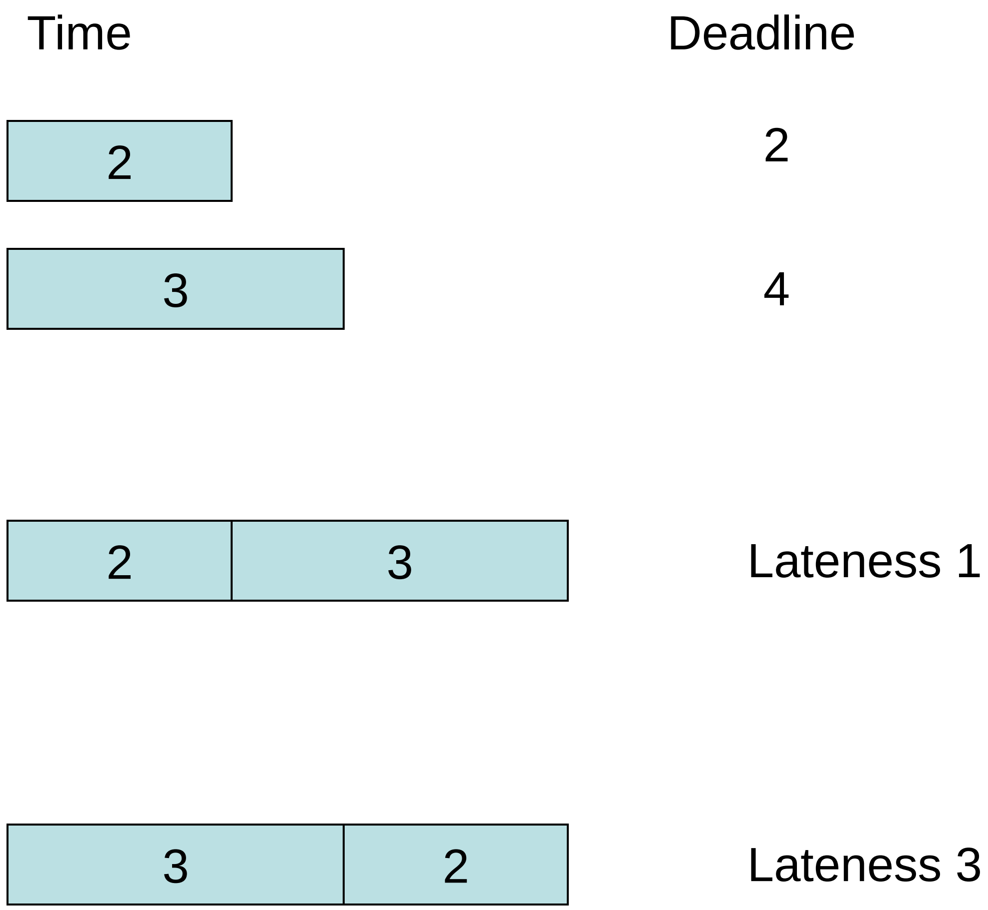
- Correctness proof: When we reach an item, we always have an open slot

Scheduling tasks

- Each task has a length t_i and a deadline d_i
- All tasks are available at the start
- One task may be worked on at a time
- All tasks must be completed

- Goal minimize maximum lateness
  - Lateness = $f_i - d_i$ if $f_i >= d_i$

Example



Determine the minimum lateness

Time	Deadline
2	6
3	4
4	5
5	12

Greedy Algorithm

- Earliest deadline first
- Order jobs by deadline

- This algorithm is optimal

Analysis

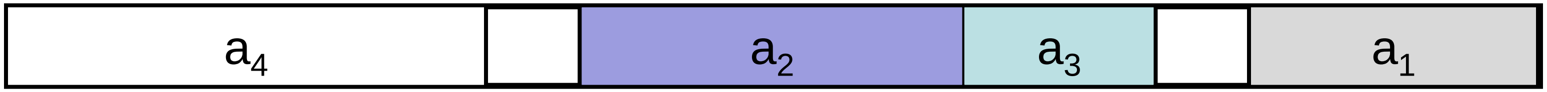
- Suppose the jobs are ordered by deadlines, $d_1 <= d_2 <= . . . <= d_n$
- A schedule has an *inversion* if job j is scheduled before i where j > i

- The schedule A computed by the greedy algorithm has no inversions.
- Let O be the optimal schedule, we want to show that A has the same maximum lateness as O

List the inversions

	Time	Deadline
a_1	3	4
a_2	4	5
a_3	2	6
a_4	5	12

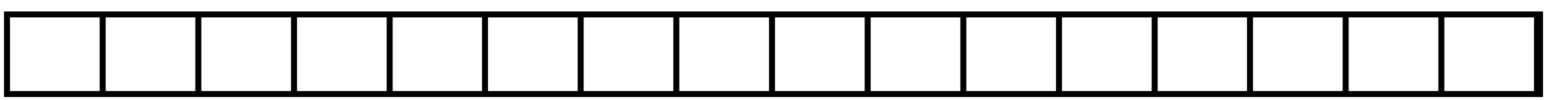
Lemma: There is an optimal schedule with no idle time



- It doesn’t hurt to start your homework early!
- Note on proof techniques
  - This type of can be important for keeping proofs clean
  - It allows us to make a simplifying assumption for the remainder of the proof

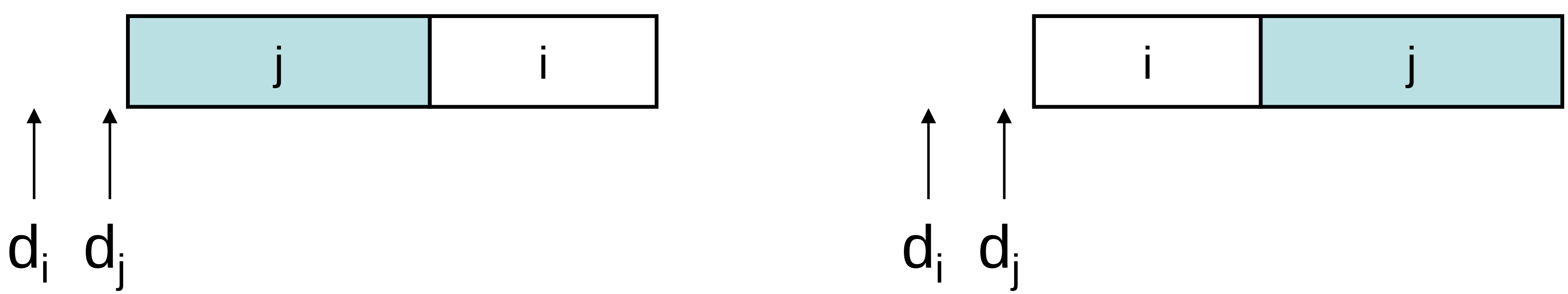
Lemma

- If there is an inversion i, j, there is a pair of adjacent jobs i’, j’ which form an inversion

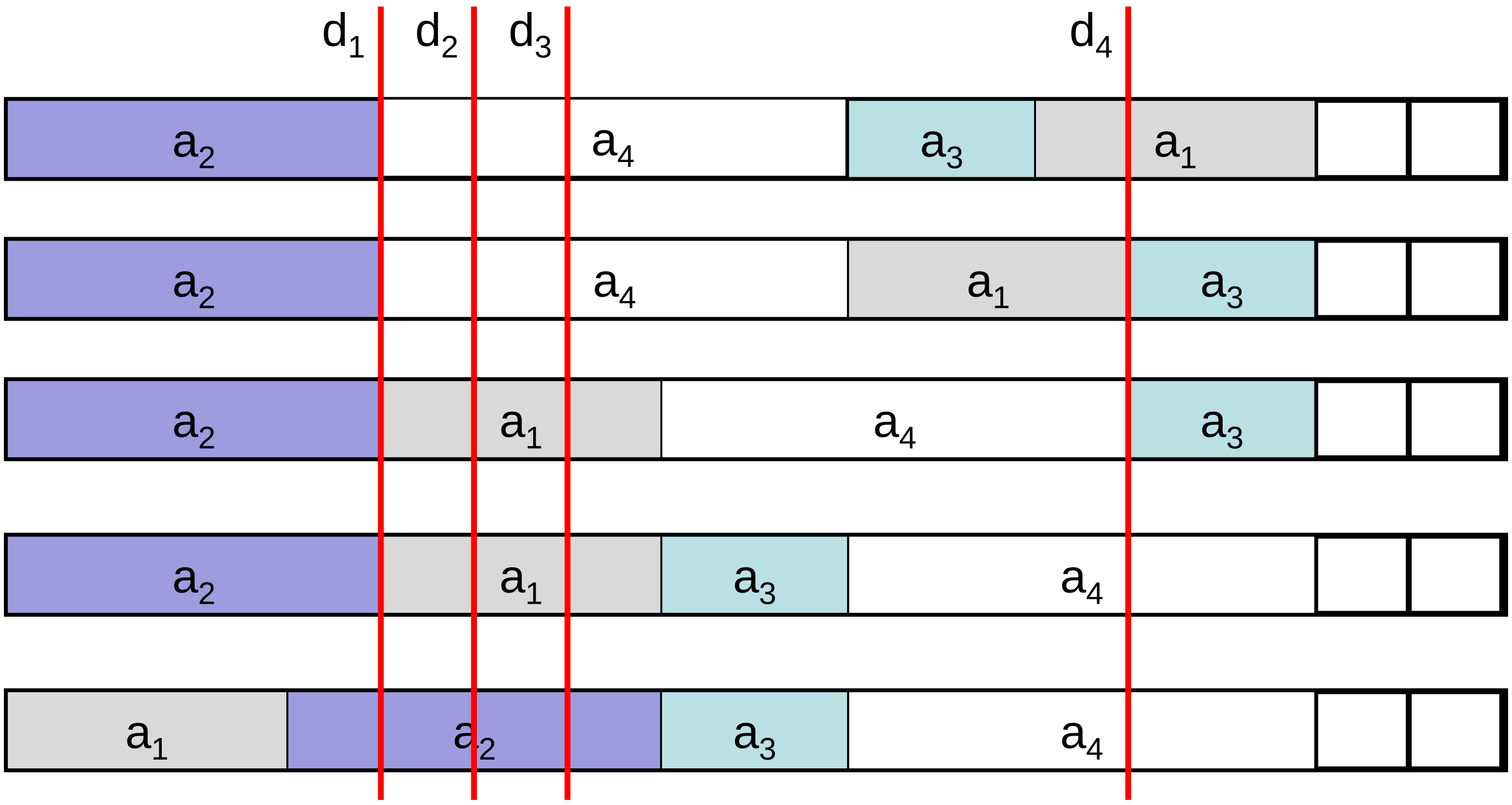


Interchange argument

- Suppose there is a pair of jobs i and j, with $d_i <= d_j$, and j scheduled immediately before i. Interchanging i and j does not increase the maximum lateness.



Proof by Bubble Sort



Determine maximum lateness

Real Proof

- There is an optimal schedule with no inversions and no idle time.
- Let O be an optimal schedule k inversions, we construct a new optimal schedule with k-1 inversions
- Repeat until we have an optimal schedule with 0 inversions
- This is the solution found by the earliest deadline first algorithm

Result

- Earliest Deadline First algorithm constructs a schedule that minimizes the maximum lateness

Homework Scheduling

- How is the model unrealistic?

Extensions

- What if the objective is to minimize the sum of the lateness?
  - EDF does not seem to work
- If the tasks have release times and deadlines, and are non-preemptable, the problem is NP-complete
- What about the case with release times and deadlines where tasks are preemptable?