EL9343

Data Structure and Algorithm

Lecture 9: Graph Basics (cont.d), Dynamic Programming

Instructor: Yong Liu

Last Lecture

- Graph Basic Definition
- Graph Representations
  - Adjacency matrix
  - Adjacency list
- Graph Traversal
  - Breath-first search
  - Depth-first search

Today

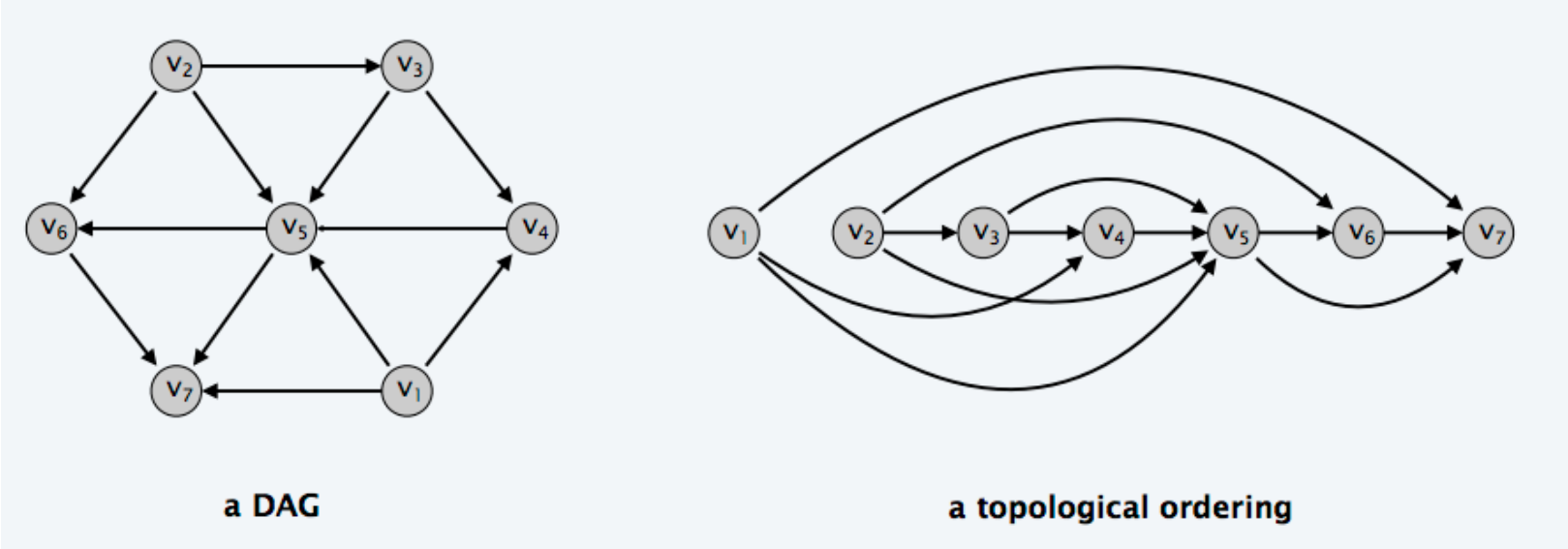
- Graphs Basics (cont.)
  - DAGs & Topological ordering
  - Strongly connected components
    - Kosaraju's algorithm
    - Tarjan's algorithm
- Introduction to Dynamic Programming

DFS & Graph Cycle: directed

- Theorem: A directed graph is *acyclic* iff a DFS yields no back edges
- Proof: (sketch, details in book)
  => DFS produces a back edge (u,v), v is an ancestor of u in depth-first tree, then in G there is a path from v to u, then back edge (u,v) completes a cycle
  <= suppose G has a cycle c, let v be the first vertex to be discovered by DFS, u is the predecessor of v in cycle c, at time d(v), all vertices of c are white, form a white path from v to u, then u becomes a descendant of v in depth-first tree, (u,v) is a back edge.

Directed Acyclic Graphs

- A DAG is a directed graph that contains no directed cycles
- A topological order/topological sort of a DAG G = (V, E) is an ordering of its nodes as v_1, v_2, …, v_n so that for every edge (v_i, v_j) we have i < j.



Precedence Constraints

- DAG's arise in many applications where there are precedence or ordering constraints.
  - e.g. If there are a series of tasks to be performed, and certain tasks must precede other tasks
- Precedence constraints.
  - Edge (v_i, v_j) means task v_i must occur before v_j.
- Applications.
  - 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.

Compute Topological Sort: Algorithm

Very easy, given DFS.

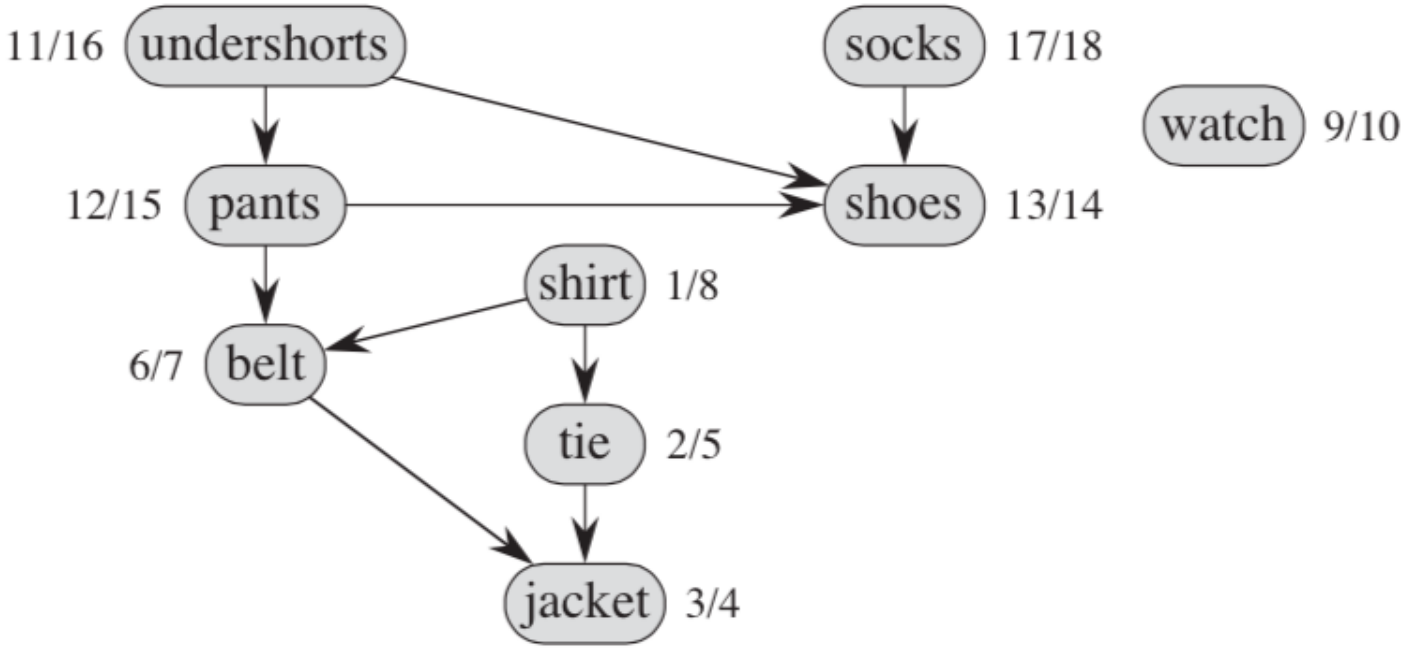
TOPOLOGICAL-SORT(G)

1. call DFS(G) to compute finishing times *f(v)* for each vertex *v*
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Compute Topological Sort: Example

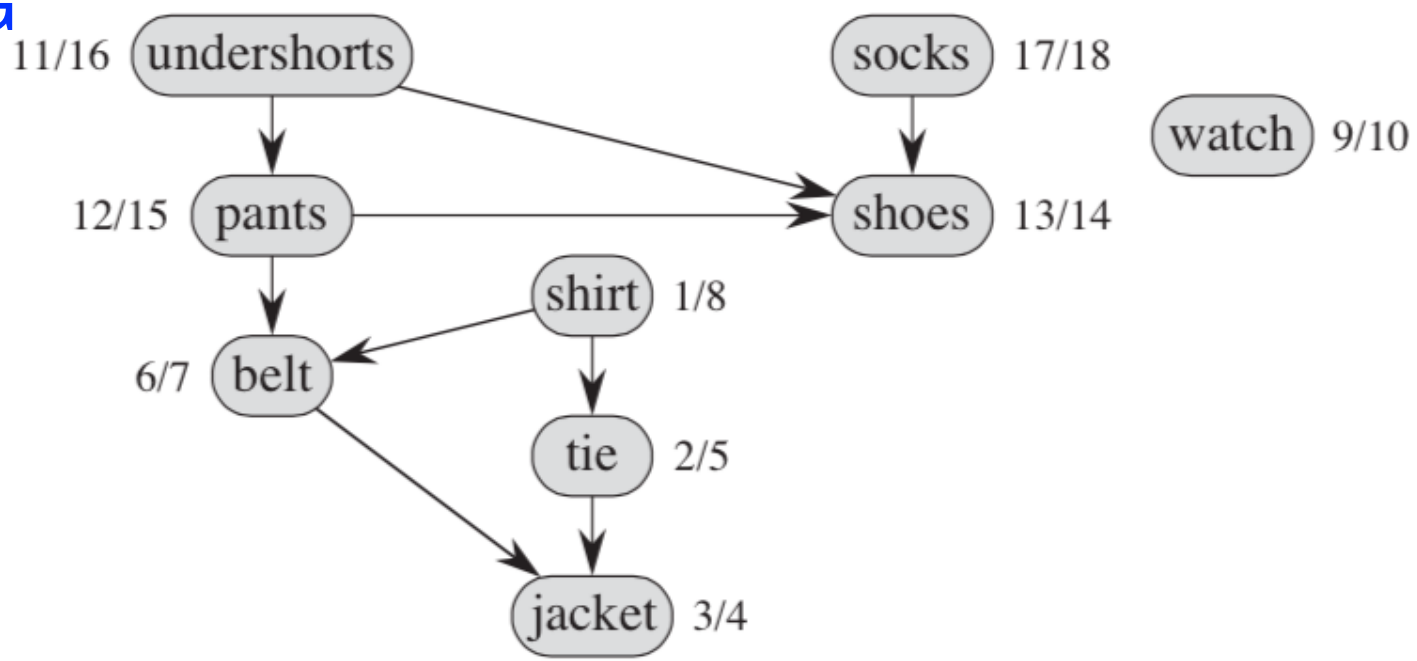
Professor Bumstead gets dressed in the morning.

- The professor must put on certain garments before others (e.g., socks before shoes).
- Other items may be put on in any order (e.g., socks and pants

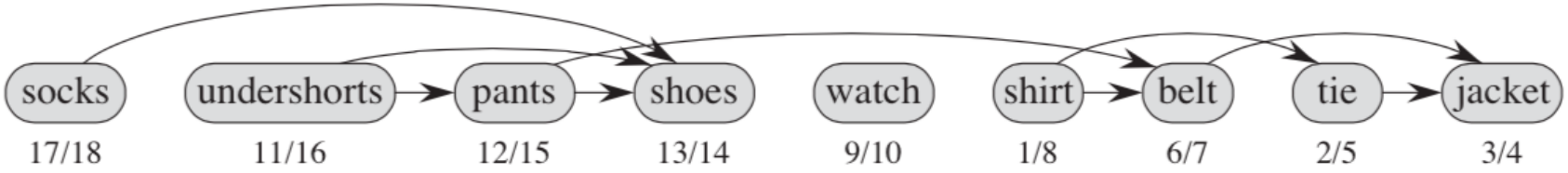


Compute Topological Sort: Example

DAG



Topological Sort



Analysis of Topological Sort

TOPOLOGICAL-SORT(G)

1. call DFS(G) to compute finishing times *f(v)* for each vertex *v*
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

- Running time for topological sort: Θ(V + E)
  - depth-first search takes Θ(V + E) time,
  - it takes O(1) time to insert each of the |V| vertices onto the front of the linked list.

Correctness of TOPOLOGICAL-SORT

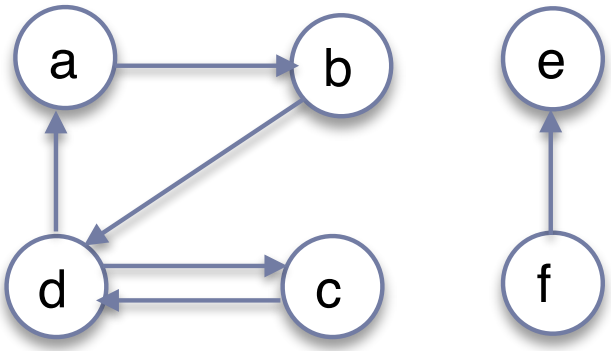
- Theorem: TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input
- Proof:
  suffices to show that if (u,v) in G, then f(u)>f(v).
  when (u,v) is explored by DFS, v cannot be gray (otherwise we have back-edge, not possible with DAG)
  1. if v is white, then v is descendant of u, f(u)>f(v)
  2. if v is black, we have finished exploring v when we are processing u (u is gray), so f(u)>f(v)

Strongly connected VS. Connected

Directed Graphs

Strongly connected: every two vertices are reachable from each other

Strongly connected components: all possible strongly connected subgraphs

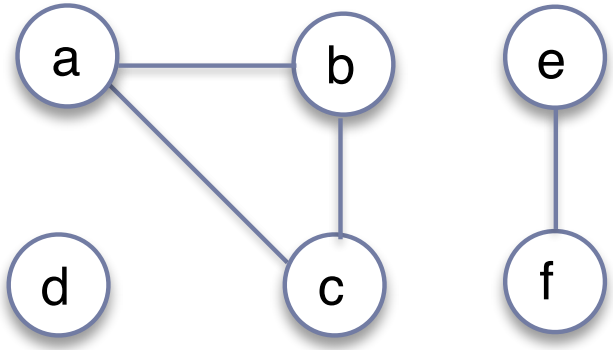


strongly connected components:
{a,b,c,d},{e},{f}

Undirected Graphs

connected: every pair of vertices are connected by a path

connected components: all possible connected subgraphs



connected components:
{a,b,c},{d},{e,f}

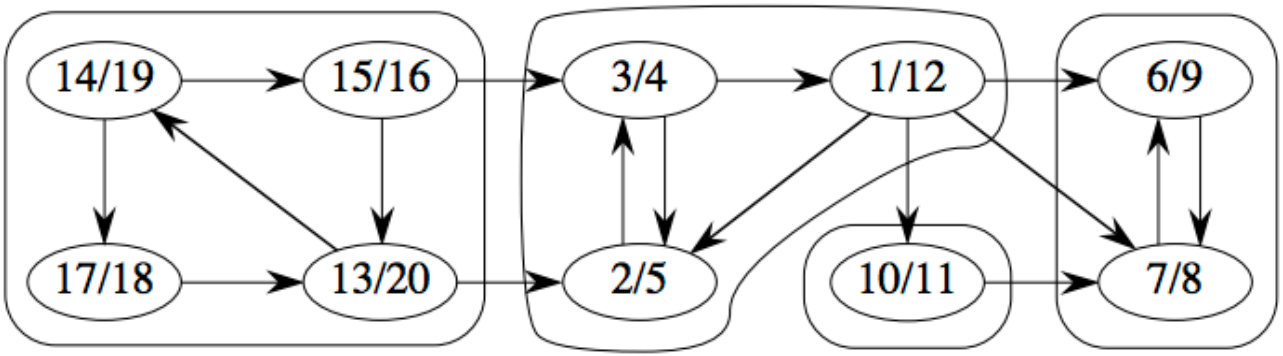
Connected Graph/Components

- how to find out whether an undirected graph is connected?

- how to find out connected components in an undirected graph?

Strongly connected components

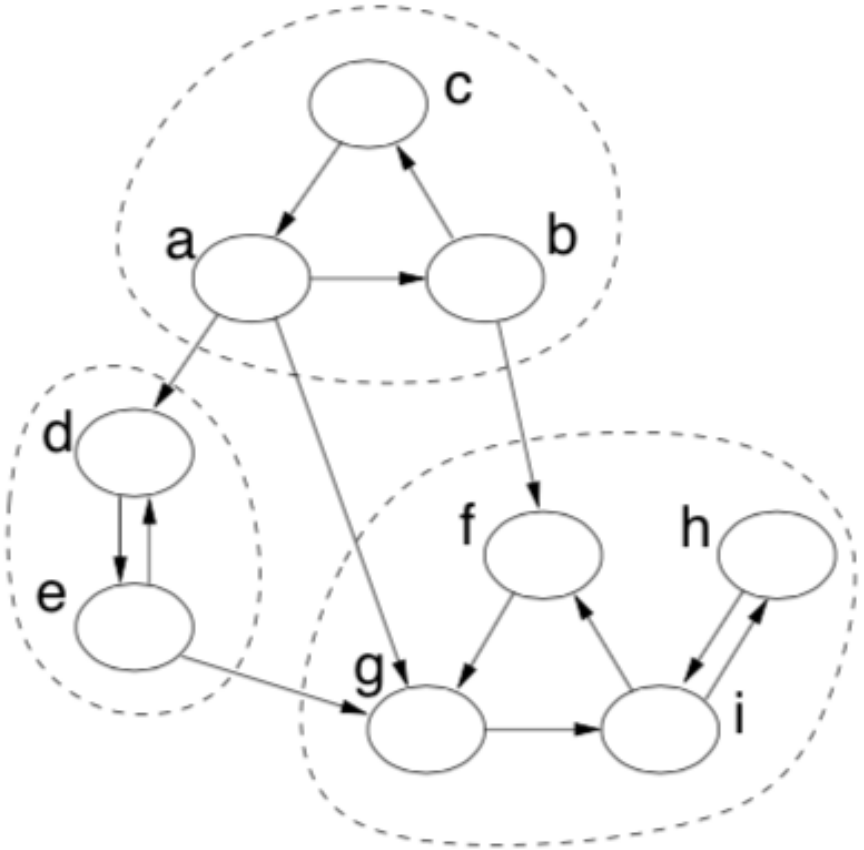
- A digraph is *strongly connected* if for every pair of vertices, u, v $\in$ V , u can reach v and vice versa.
- Important connectivity problem with digraphs.
  - E.g.,When digraphs are used in communication and transportation networks, people want to know that there networks are complete: from any location it is possible to reach any other location in the digraph.



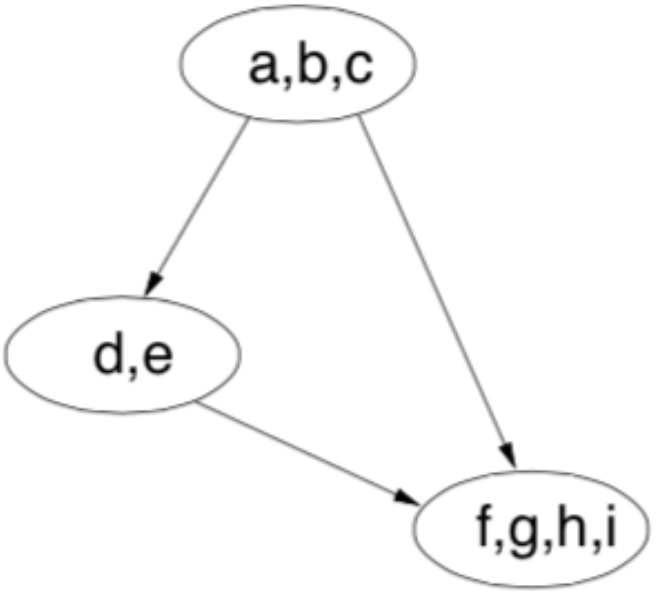
Component DAG

- Computing the *strongly connected components* of a digraph.
- Partition the vertices of the digraph into subsets such that the induced subgraph of each subset is strongly connected. (These subsets should be as large as possible, and still have this property.)
- Merging the vertices in each strong component into a single *super vertex*, and joint two super-vertices (A, B) if and only if there are vertices $u \in A$ and $v \in B$ such that $(u, v) \in E$
- The resulting digraph, called the *component digraph*, is necessarily acyclic. We may refer it as the *component DAG*.

Component DAG: Example



Digraph and Strong Components



Component DAG

Finding Strongly Connected Component

- Kosaraju's algorithm
  - based on two pass DFS
- Tarjan's algorithm
  - based on one pass revised DFS

Kosaraju's algorithm

- Kosaraju’s algorithm is amazingly simple.

- It uses a very clever trick: If you reverse all of the edges in a graph, the resulting graph has the same strongly connected components as the original.

- We can get the SCCs by doing a forward traversal to find an ordering of vertices, then doing a traversal of the reverse of the graph in the order generated by the first traversal.

Kosaraju's algorithm

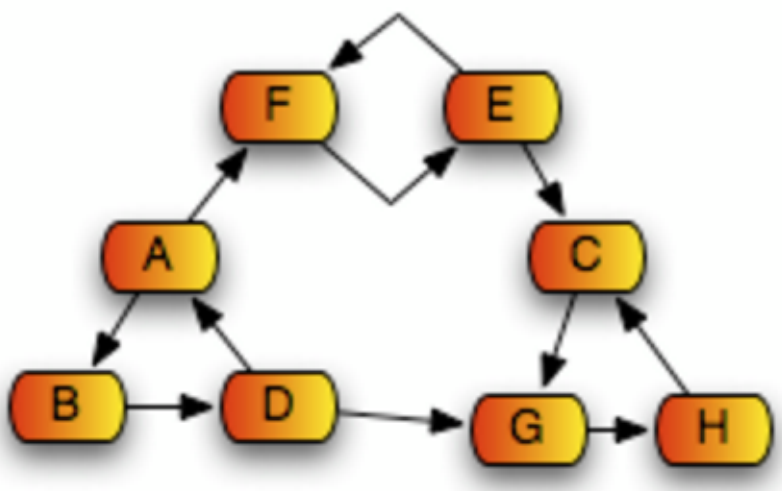
STRONGLY-CONNECTED-COMPONENTS(G)

1. call DFS(G) to compute finishing times f(u) for each vertex u
2. compute G^T (reversing all edges of G)
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing f(u) (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

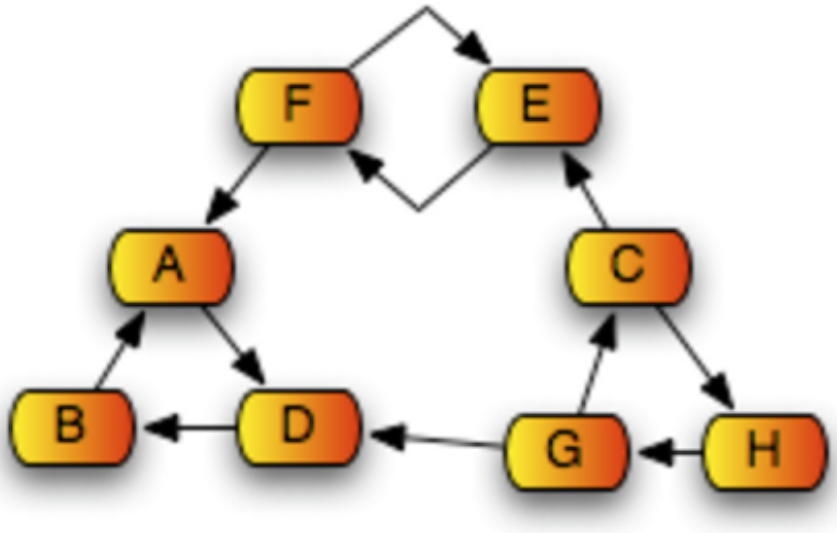
Running time: Θ(V + E)

Kosaraju's algorithm: Example

G



G^T



SCCs: {A, B, D}, {E, F}, and {C, G, H}

Correctness of Kosaraju's algorithm

Proof Sketch:

1. View G as Component DAG
2. After first DFS, each component (C) has a finish time f(C), which is the latest finish time of all nodes in C;
3. If there is an edge (u,v) from C' to C, then f(C) < f(C') (because there will be no path from C back to C')
4. The second DFS pass applies to G^T , which has the same connected components as G. DFS starts with the node v with largest f(v), i.e., it starts with the component C_1 with largest f(C_1), then there is no edge from any other component C' to C_1 in G (otherwise f(C_1) < f(C')), i.e., there is no edge from C_1 to any other component C' in G^T
5. DFS will return the first DFS tree for nodes in C_1
6. DFS will move to component C_2 with the second largest f(C_2) value, similar arguments applies, to show that it will return a DFS tree for nodes in C_2
7. Repeat until all components are traversed by DFS

Tarjan's algorithm

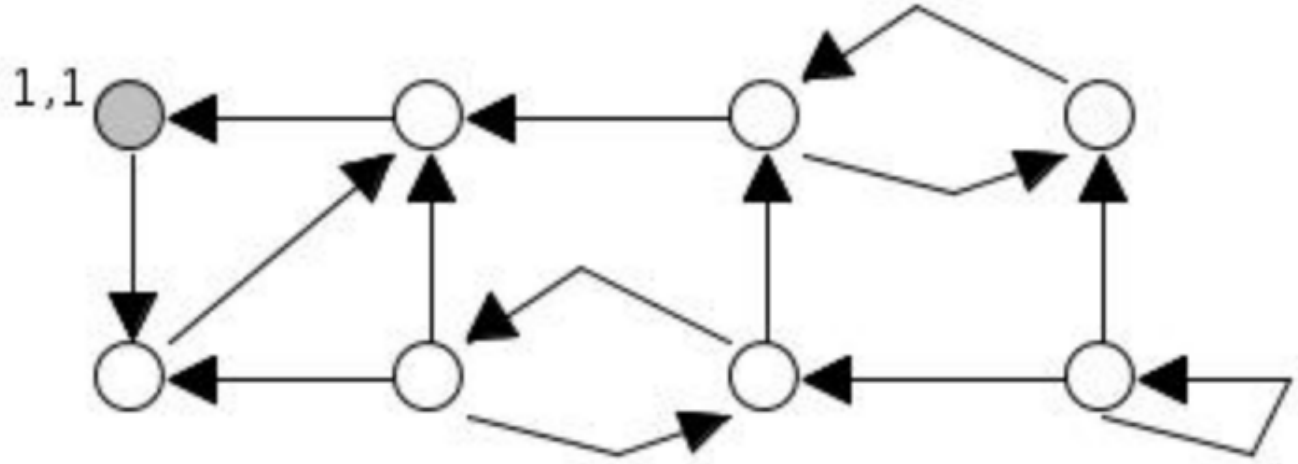
- Tarjan Algorithm is based on following facts:
- 1. DFS search produces a DFS tree/forest
- 2. Strongly Connected Components form subtrees of the DFS tree.
- 3. If we can find head of such subtrees, we can print/ store all the nodes in that subtree (including head) and that will be one SCC.
- 4. There is no back edge from one SCC to another

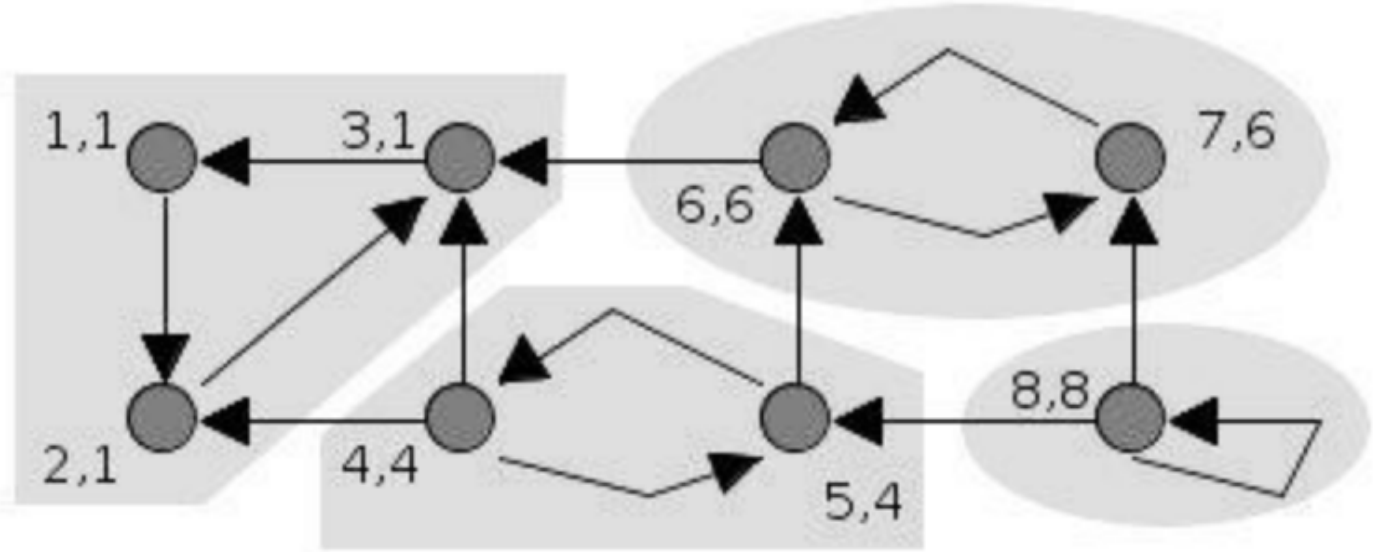
Tarjan's algorithm

- The vertices are indexed as they are traversed by DFS procedure.
- While returning from the recursion of DFS, every vertex v gets assigned a vertex L as a representative.
- L is a vertex with the least index that can be reach from v.
- Nodes with the same representative assigned are located in the same strongly connected component.

```
DFSVisit(u) {
    color[u] = gray;
    d[u] = ++time;
    for each v in Adj(u) do
        if (color[v] == white) {
            pred[v] = u;
            DFSVisit(v);
        }
    color[u] = black;
    f[u] = ++time;
}
```

Tarjan's algorithm: Example





Analysis of Tarjan's algorithm

Running time: Θ(V + E)

- One pass modified DFS

Pseudocode and more details: check on wiki page:

- https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

Dynamic Programming

- Dynamic programming: Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

- An algorithm design technique (like divide and conquer)

- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

Dynamic Programming

- Applicable when subproblems are **not** independent
  - Subproblems share sub-subproblems
- A divide and conquer approach would repeatedly solve the common subproblems
- Dynamic programming solves every subproblem just once and stores the answer in a table

Dynamic Programming

- Used for **optimization problems**

  e.g. "find XX with the smallest/largest YY" (for two strings, find a longest common subsequence; for a set of strings with frequencies, find the best way to organize them into a search tree to minimize the expected cost; etc).

- A set of choices must be made to get an optimal solution
- Find a solution with the optimal value (minimum or maximum)
- There may be many solutions that lead to an optimal value
- Our goal: **find an optimal solution**

Dynamic Programming Applications

Areas

- Bioinformatics.
- Control theory.
- Computer science: theory, graphics, AI, compilers, systems, …
- ...

Some famous dynamic programming algorithms

- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- ...

Dynamic Programming Algorithm

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution, typically in a bottom-up fashion
4. **Construct** an optimal solution from computed information (not always necessary)

Fibonacci Again!

- Let us consider the problem to find the nth fibonacci number.

  1 1 2 3 5 8 13 21 34 44 89 ...

- Here's the infamous recursive algorithm

  Algorithm Fibonacci(n)

  if n <= 1, then:

  return 1

  else:

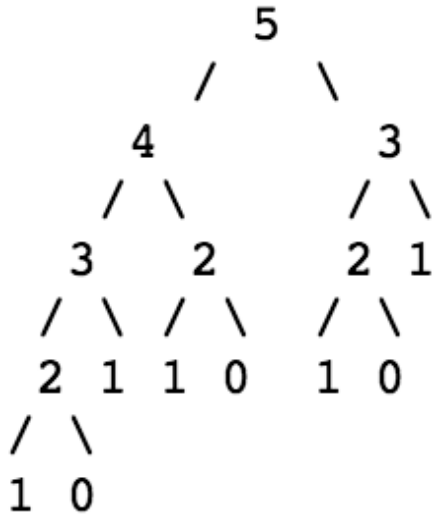
  return Fibonacci(n - 1) + Fibonacci(n - 2)

  end of if

Recursive Algorithm

- To compute Fib(5), we are calculating Fib(3) twice, Fib(2) three times, and Fib(1) five times.
- Imagine calculating 100th Fibonacci number, or millionth! Repeating the exact same computation many times is **terribly slow!**
- **This takes $O(1.618^n)$ time!**

to compute	takes
Fibonacci(40)	75.22 seconds
Fibonacci(70)	4.43 years

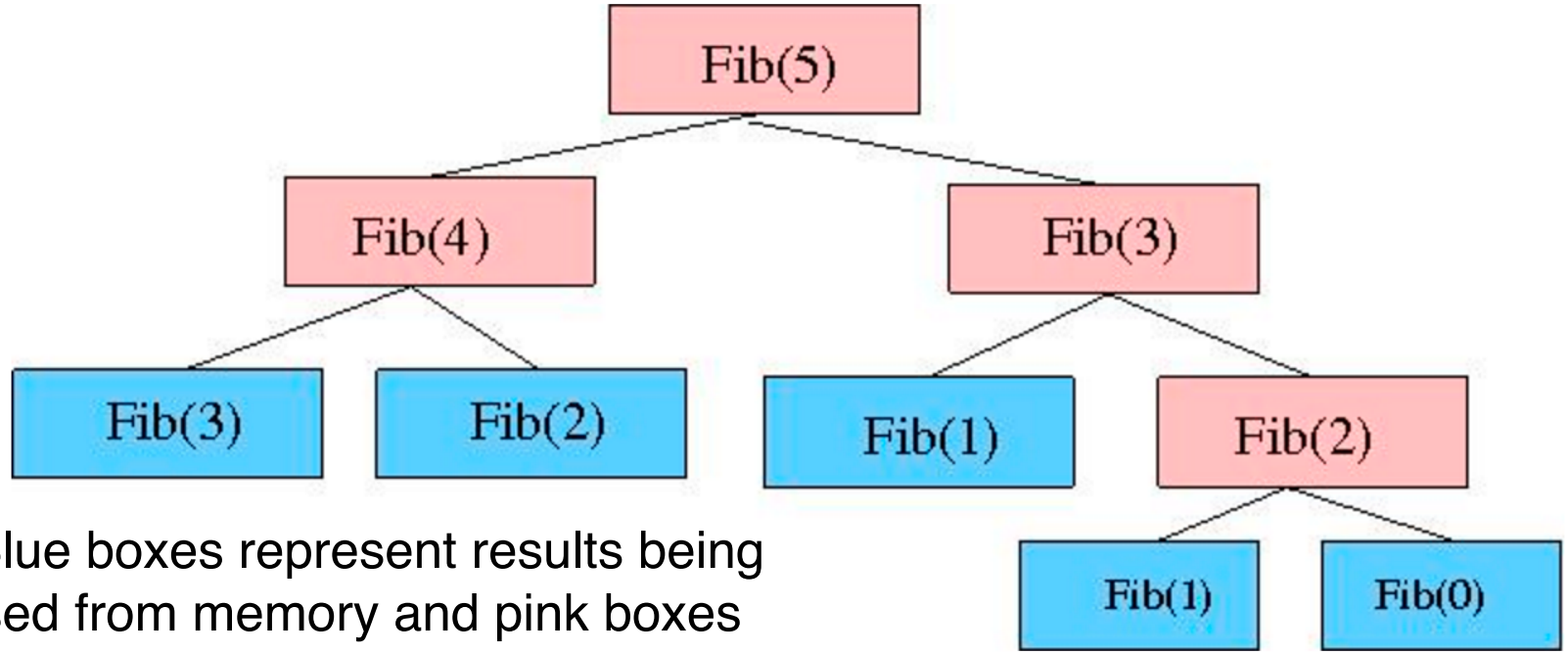


Memorization Instead of Repeating Calculation

- We can save all this effort by computing only once and re-using every other time.
  - Storing the results the first time we compute them
  - For later, instead of doing the same computation again, just look it up from the stored memory.

- This technique of avoiding repeated calculation of results by storing them is called memoization and is used in Dynamic Programming.

The Dynamic Programming Approach



*Blue boxes represent results being used from memory and pink boxes represent results being computed.

- **This takes O(n) time. From exponential to linear!**

to compute	took	now takes
Fibonacci(40)	75.22 sec	2 microseconds
Fibonacci(70)	4.43 years	3 microseconds

The Dynamic Programming Approach

- Dynamic programming suggests we start at the bottom and work up.

  Algorithm Fast-Fibonacci(n)

  Let fib[0] and fib[1] be 1.

  for each i from 2 to n, do:

  Let fib[i] be fib[i - 2] + fib[i - 1].

  end of loop

  return fib[n].

Rod Cutting

- Solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

Rod Cutting

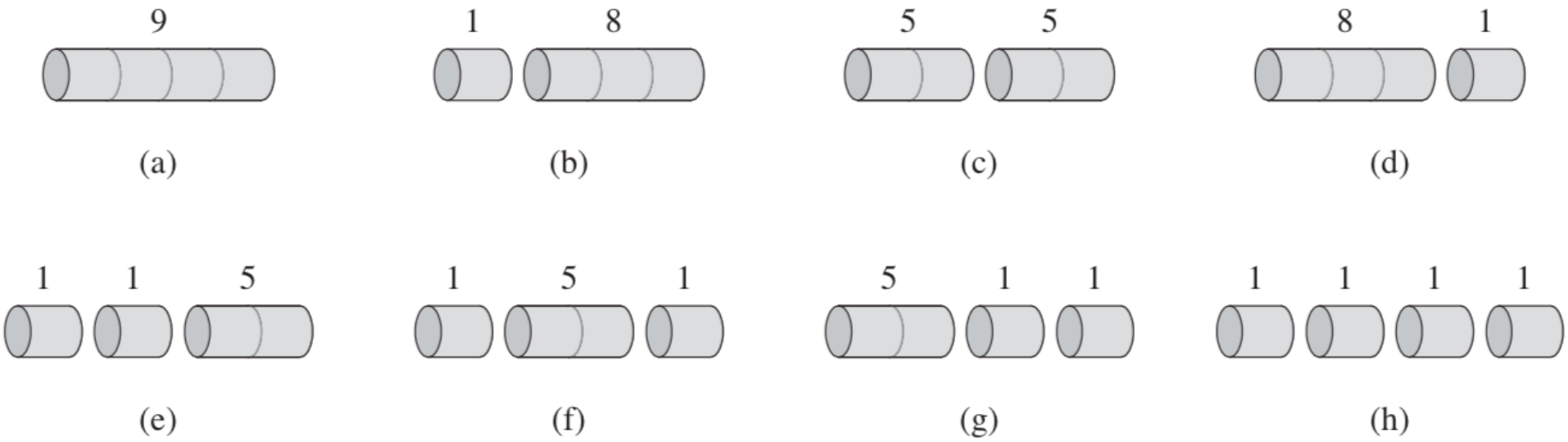
- We assume that we know, for i=1, 2, ..., the price p_i in dollars that Serling Enterprises charges for a rod of length i inches. Rod lengths are always an integral number of inches.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for i=1, 2, ...,n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Rod Cutting

- Consider the case when n=4.



The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

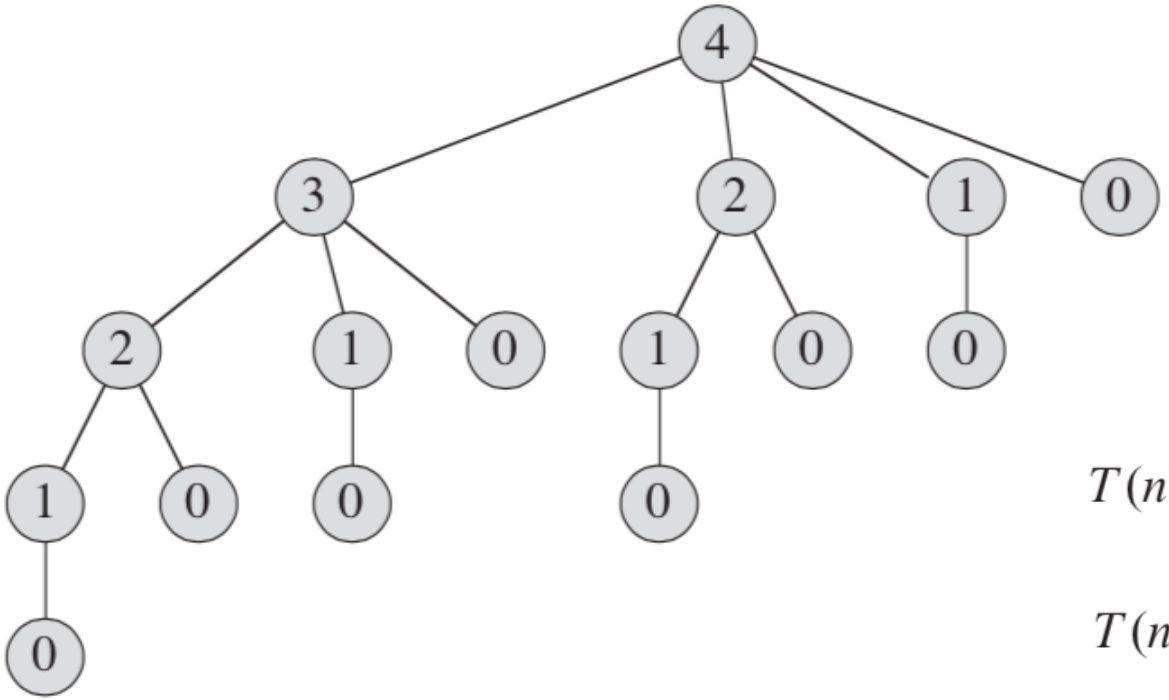
Rod Cutting

- How about more general case n?

Recursive top-down implementation

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Recursive top-down implementation



$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) .$$

$$T(n) = 2^n$$

The recursion tree showing recursive calls resulting from a call CUT-ROD(p,n) for n=4. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size s-t and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

Using dynamic programming for optimal rod cutting

- Top-down with memoization

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

Running time $\Theta(n^2)$

Using dynamic programming for optimal rod cutting

- Bottom-up dynamic-programming approach

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

Running time $\Theta(n^2)$

Reconstructing a solution

- Above dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes.

- We can extend the dynamic-programming approach to record not only the optimal *value* computed for each subproblem, but also a *choice* that led to the optimal value. With this information, we can readily print an optimal solution.

Reconstructing a solution

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

Reconstructing a solution

In this rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD(p,10) would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Next Lecture

- More Dynamic Programming Examples
  - Longest common subsequence
  - Optimal binary search tree

- Introduction to greedy algorithm