Greedy Algorithm, Elementary Graph Algorithms

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Nov 21 & 23, 2017

Greedy Algorithm (Chap 16)

Similar to dynamic programming.

Used for optimization problems.

Idea: When we have a choice to make, make the one that looks best *right now*. Make a *locally optimal choice* in hope of getting a *globally optimal solution*.

Greedy algorithms don't always yield an optimal solution. But sometimes they do. We'll see a problem for which they do. Then we'll look at some general characteristics of when greedy algorithms give optimal solutions.

Activity Selection

n ***activities*** require *exclusive* use of a common resource. For example, scheduling the use of a classroom.

Set of activities $S = \{a_1, \ldots, a_n\}$.

a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i = start time and f_i = finish time.

Goal: Select the largest possible set of nonoverlapping (***mutually compatible***) activities.

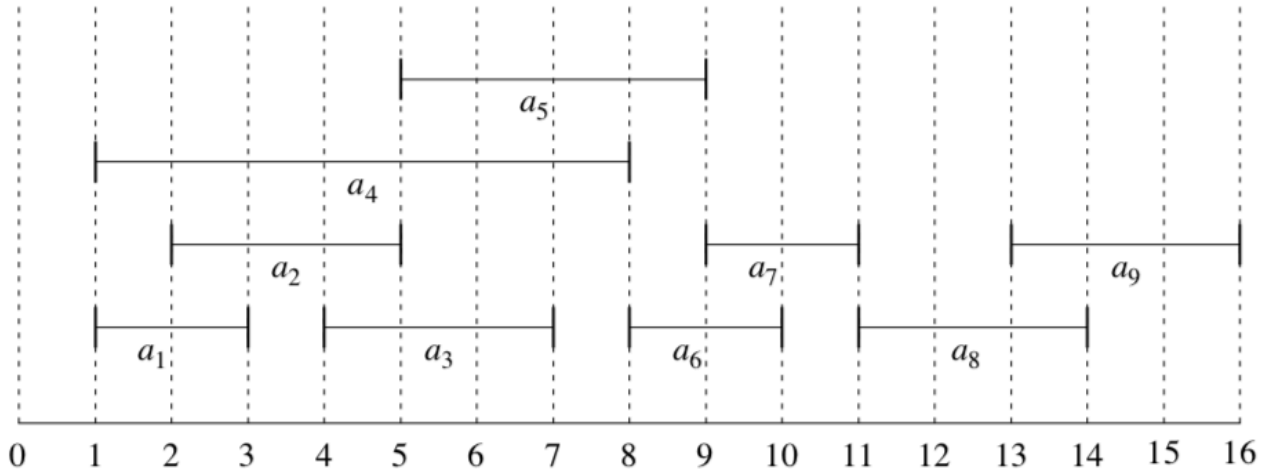
Note: Could have many other objectives:

- Schedule room for longest time.
- Maximize income rental fees.

Activity Example

Example: S sorted by finish time:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



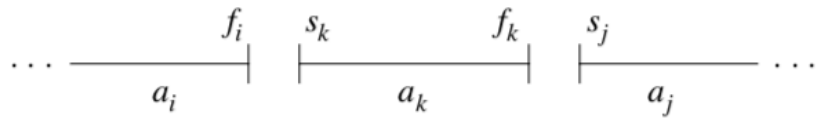
Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

Not unique: also $\{a_2, a_5, a_7, a_9\}$.

Optimal Substructure

Optimal substructure of activity selection

$$\begin{aligned} S_{ij} &= \{a_k \in S : f_i \le s_k < f_k \le s_j\} \\ &= \text{activities that start after } a_i \text{ finishes and finish before } a_j \text{ starts .} \end{aligned}$$



Activities in S_{ij} are compatible with

- all activities that finish by f_i, and
- all activities that start no earlier than s_j.

To represent the entire problem, add fictitious activities:

$a_0 = [-\infty, 0)$

$a_{n+1} = [\infty, \text{“}\infty + 1\text{”})$

We don't care about $-\infty$ in a_0 or “$\infty + 1$” in a_{n+1}.

Then $S = S_{0,n+1}$.

Range for S_{ij} is $0 \le i, j \le n + 1$.

Optimal Substructure

Assume that activities are sorted by monotonically increasing finish time:

$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1}$.

Then $i \geq j \Rightarrow S_{ij} = \emptyset$.

- If there exists $a_k \in S_{ij}$:

  $f_i \leq s_k < f_k \leq s_j < f_j \Rightarrow f_i < f_j$.

- But $i \geq j \Rightarrow f_i \geq f_j$. Contradiction.

So only need to worry about S_{ij} with $0 \leq i < j \leq n + 1$.

All other S_{ij} are $\emptyset$.

Suppose that a solution to S_{ij} includes a_k. Have 2 subproblems:

- S_{ik} (start after a_i finishes, finish before a_k starts)
- S_{kj} (start after a_k finishes, finish before a_j starts)

Optimal Substructure

Solution to S_{ij} is (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj}).

Since a_k is in neither subproblem, and the subproblems are disjoint,

$$|\text{solution to } S| = |\text{solution to } S_{ik}| + 1 + |\text{solution to } S_{kj}| \ .$$

If an optimal solution to S_{ij} includes a_k, then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well. Use the usual cut-and-paste argument.

Let $A_{ij} =$ optimal solution to S_{ij}.

So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ *[leave on board]* , assuming:

- S_{ij} is nonempty, and
- we know a_k.

Recursive solution

Recursive solution to activity selection

$c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_j.

- $i \geq j \Rightarrow S_{ij} = \emptyset \Rightarrow c[i, j] = 0$.

If $S_{ij} \neq \emptyset$, suppose we know that a_k is in the subset. Then

$$c[i, j] = c[i, k] + 1 + c[k, j] .$$

But of course we don't know which k to use, and so

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset . \end{cases}$$

Recursive solution

Theorem

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min\{f_k : a_k \in S_{ij}\}$. Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

Recursive solution

Proof

2. Suppose there is some $a_k \in S_{im}$. Then $f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$. Then $a_k \in S_{ij}$ and it has an earlier finish time than f_m, which contradicts our choice of a_m. Therefore, there is no $a_k \in S_{im} \Rightarrow S_{im} = \emptyset$.
1. Let A_{ij} be a maximum-size subset of mutually compatible activities in S_{ij}.

   Order activites in A_{ij} in monotonically increasing order of finish time.

   Let a_k be the first activity in A_{ij}.

   If $a_k = a_m$, done (a_m is used in a maximum-size subset).

   Otherwise, construct $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m).

Claim

Activities in A'_{ij} are disjoint.

Proof Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish, $f_m \leq f_k$ (so a_m doesn't overlap anything else in A'_{ij}). ■ (claim)

Since $\left|A'_{ij}\right| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is A'_{ij}. ■ (theorem)

Recursive solution

This is great:

	before theorem	after theorem
# of subproblems in optimal solution	2	1
# of choices to consider	$j - i - 1$	1

Now we can solve *top down*:

- To solve a problem S_{ij},
  - Choose $a_m \in S_{ij}$ with earliest finish time: the ***greedy choice***.
  - Then solve S_{mj}.

What are the subproblems?

- Original problem is $S_{0,n+1}$.
- Suppose our first choice is a_{m_1}.
- Then next subproblem is $S_{m_1,n+1}$.
- Suppose next choice is a_{m_2}.
- Next subproblem is $S_{m_2,n+1}$.
- And so on.

Each subproblem is $S_{m_i,n+1}$, i.e., the last activities to finish.

And the subproblems chosen have finish times that increase.

Therefore, we can consider each activity just once, in monotonically increasing order of finish time.

Recursive solution

Easy recursive algorithm: Assumes activites already sorted by monotonically increasing finish time. (If not, then sort in $O(n \lg n)$ time.) Return an optimal solution for $S_{i,n+1}$:

REC-ACTIVITY-SELECTOR(s, f, i, n)
$m \leftarrow i + 1$
while $m \leq n$ and $s_m < f_i$ ▷ Find first activity in $S_{i,n+1}$.
  do $m \leftarrow m + 1$
if $m \leq n$
  then return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
  else return $\emptyset$

Initial call: REC-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Recursive solution

Idea: The **while** loop checks $a_{i+1}, a_{i+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m,n+1}$, and return this solution, along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

Go through example given earlier. Should get $\{a_1, a_4, a_8, a_{11}\}$.

Time: $\Theta(n)$—each activity examined exactly once.

Iterative solution

Can make this *iterative*. It's already almost tail recursive.

GREEDY-ACTIVITY-SELECTOR(s, f, n)

$A \leftarrow \{a_1\}$
$i \leftarrow 1$
for $m \leftarrow 2$ **to** n
    do if $s_m \geq f_i$
        then $A \leftarrow A \cup \{a_m\}$
            $i \leftarrow m$ $\quad \triangleright$ a_i is most recent addition to A
return A

Go through example given earlier. Should again get $\{a_1, a_4, a_8, a_{11}\}$.

Time: $\Theta(n)$.

Greedy strategy

The choice that seems best at the moment is the one we go with.

What did we do for activity selection?

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Prove that at any stage of recursion, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
4. Show that all but one of the subproblems resulting from the greedy choice are empty.
5. Develop a recursive greedy algorithm.
6. Convert it to an iterative algorithm.

Greedy Algorithm

At first, it looked like dynamic programming.

Typically, we streamline these steps.

Develop the substructure with an eye toward

- making the greedy choice,
- leaving just one subproblem.

For activity selection, we showed that the greedy choice implied that in S_j, only i varied, and j was fixed at $n + 1$.

We could have started out with a greedy algorithm in mind:

- Define $S_i = \{a_k \in S : f_i \leq s_k\}$.
- Then show that the greedy choice—first a_m to finish in S_i—combined with optimal solution to $S_m \Rightarrow$ optimal solution to S_i.

Greedy Algorithm

Typical streamlined steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.

No general way to tell if a greedy algorithm is optimal, but two key ingredients are

1. greedy-choice property and
2. optimal substructure.

Greedy-choice property

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

Dynamic programming:

- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
- Solve *bottom-up*.

Greedy Algorithm

Greedy:

- Make a choice at each step.
- Make the choice *before* solving the subproblems.
- Solve *top-down*.

Typically show the greedy-choice property by what we did for activity selection:

- Look at a globally optimal solution.
- If it includes the greedy choice, done.
- Else, modify it to include the greedy choice, yielding another solution that's just as good.

Can get efficiency gains from greedy-choice property.

- Preprocess input to put it into greedy order.
- Or, if dynamic data, use a priority queue.

Optimal substructure

Just show that optimal solution to subproblem and greedy choice $\Rightarrow$ optimal solution to problem.

Greedy vs. dynamic programming

The knapsack problem is a good example of the difference.

0-1 knapsack problem:

- n items.
- Item i is worth $\$v_i$, weighs w_i pounds.
- Find a most valuable subset of items with total weight $\leq W$.
- Have to either take an item or not take it—can't take part of it.

Greedy Algorithm

Fractional knapsack problem: Like the 0-1 knapsack problem, but can take fraction of an item.

Both have optimal substructure.

But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.

To solve the fractional problem, rank items by value/weight: v_i/w_i.

Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i.

FRACTIONAL-KNAPSACK(v, w, W)

$load \leftarrow 0$
$i \leftarrow 1$
while $load < W$ and $i \leq n$
    do if $w_i \leq W - load$
        then take all of item i
        else take $(W - load)/w_i$ of item i
      add what was taken to $load$
      $i \leftarrow i + 1$

Greedy Algorithm

Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

Greedy doesn't work for the 0-1 knapsack problem. Might get empty space, which lowers the average value per pound of the items taken.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50$.

Greedy solution:

- Take items 1 and 2.
- value $= 160$, weight $= 30$.

Have 20 pounds of capacity left over.

Optimal solution:

- Take items 2 and 3.
- value $= 220$, weight $= 50$.

No leftover capacity.

Elementary Graph Algorithms (Chap 22)

Given graph $G = (V, E)$.

- May be either directed or undirected.
- Two common ways to represent for algorithms:
  1. Adjacency lists.
  2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$. In asymptotic notation—and *only* in asymptotic notation—we'll drop the cardinality. Example: $O(V + E)$.

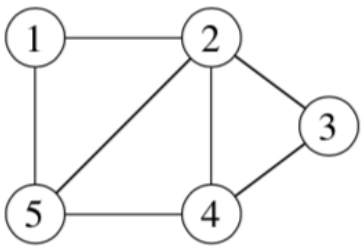
Adjacency lists

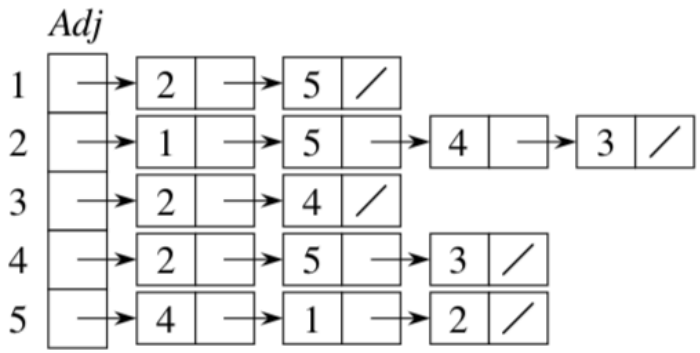
Adjacency lists

Array Adj of $|V|$ lists, one per vertex.

Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

Example: For an undirected graph:





Directed case

If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbf{R}$

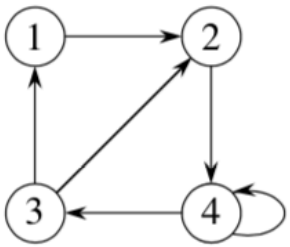
We'll use weights later on for spanning trees and shortest paths.

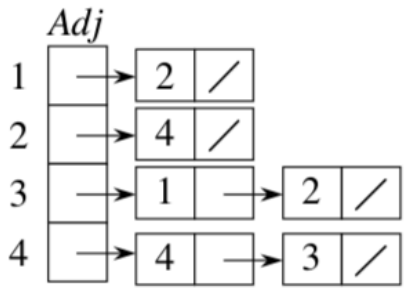
Space: $\Theta(V + E)$.

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine if $(u, v) \in E$: $O(\text{degree}(u))$.

Example: For a directed graph:





Same asymptotic space and time.

Adjacency matrix

Adjacency matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine if $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

We'll use both representations in these lecture notes.

Breadth First Search

Input: Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.

Output: $d[v]$ = distance (smallest # of edges) from s to v, for all $v \in V$.
In book, also $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.

- u is v's ***predecessor***.
- set of edges $\{(\pi[v], v) : v \neq s\}$ forms a tree.

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple.

- Compute only $d[v]$, not $\pi[v]$. *[See book for $\pi[v]$.]*
- Omitting colors of vertices. *[Used in book to reason about the algorithm. We'll skip them here.]*

BFS

Idea: Send a wave out from s.

- First hits all vertices 1 edge from s.
- From there, hits all vertices 2 edges from s.
- Etc.

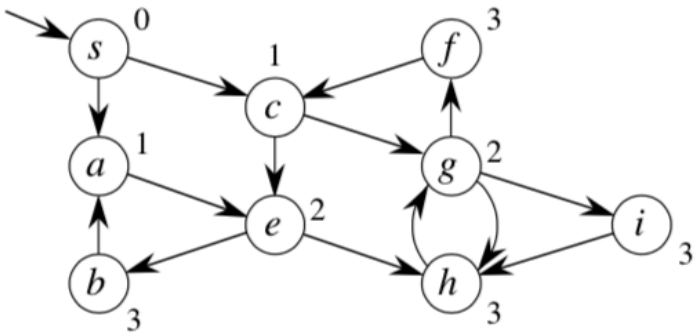
Use FIFO queue Q to maintain wavefront.

- $v \in Q$ if and only if wave has hit v but has not come out of v yet.

```
BFS(V, E, s)
for each u ∈ V − {s}
    do d[u] ← ∞
d[s] ← 0
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ Adj[u]
           do if d[v] = ∞
                 then d[v] ← d[u] + 1
                      ENQUEUE(Q, v)
```

Example

Example: directed graph *[undirected example in book]* .



Can show that Q consists of vertices with d values.

$i \quad i \quad i \quad \ldots \quad i \quad i+1 \quad i+1 \quad \ldots \quad i+1$

- Only 1 or 2 values.
- If 2, differ by 1 and all smallest are first.

BFS Analysis

Since each vertex gets a finite d value at most once, values assigned to vertices are monotonically increasing over time.

Actual proof of correctness is a bit trickier. See book.

BFS may not reach all vertices.

Time $= O(V + E)$.

- $O(V)$ because every vertex enqueued at most once.
- $O(E)$ because every vertex dequeued at most once and we examine (u, v) only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

Depth First Search

Input: $G = (V, E)$, directed or undirected. No source vertex given!

Output: 2 ***timestamps*** on each vertex:

- $d[v] =$ ***discovery time***
- $f[v] =$ ***finishing time***

These will be useful for other algorithms later on.

Can also compute $\pi[v]$. *[See book.]*

Will methodically explore *every* edge.

- Start over from different vertices as necessary.

As soon as we discover a vertex, explore from it.

- Unlike BFS, which puts a vertex on a queue so that we explore from it later.

As DFS progresses, every vertex has a ***color***:

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)

Discovery and finish times:

- Unique integers from 1 to $2|V|$.
- For all v, $d[v] < f[v]$.

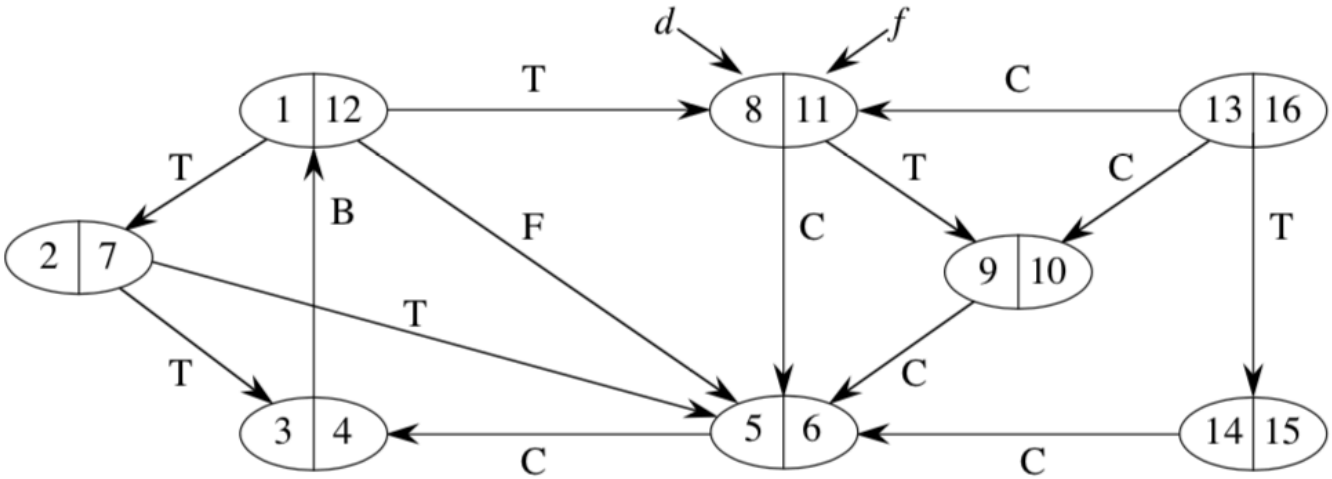
In other words, $1 \leq d[v] < f[v] \leq 2|V|$.

Pseudocode: Uses a global timestamp *time*.

```
DFS(V, E)
for each u ∈ V
    do color[u] ← WHITE
time ← 0
for each u ∈ V
    do if color[u] = WHITE
          then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY          ▷ discover u
time ← time +1
d[u] ← time
for each v ∈ Adj[u]      ▷ explore (u, v)
    do if color[v] = WHITE
          then DFS-VISIT(v)
color[u] ← BLACK
time ← time +1
f[u] ← time              ▷ finish u
```

Example



Time $= \Theta(V + E)$.

- Similar to BFS analysis.
- Θ, not just O, since guaranteed to examine every vertex and edge.

DFS forms a ***depth-first forest*** comprised of > 1 ***depth-first trees***. Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

Parenthesis Theorem

Theorem (Parenthesis theorem)

[Proof omitted.]

For all u, v, exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither of u and v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.

Like parentheses:

- OK: ( ) [ ] ([ ]) [( )]
- Not OK: ([ )] [( ])

White-path Theorem

Corollary

v is a proper descendant of u if and only if $d[u] < d[v] < f[v] < f[u]$.

Theorem (White-path theorem)

[Proof omitted.]

v is a descendant of u if and only if at time $d[u]$, there is a path $u \rightsquigarrow v$ consisting of only white vertices. (Except for u, which was *just* colored gray.)

Edge classes

Classification of edges

- ***Tree edge:*** in the depth-first forest. Found by exploring (u, v).
- ***Back edge:*** (u, v), where u is a descendant of v.
- ***Forward edge:*** (u, v), where v is a descendant of u, but not a tree edge.
- ***Cross edge:*** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

[Now label the example from above with edge types.]

In an undirected graph, there may be some ambiguity since (u, v) and (v, u) are the same edge. Classify by the first type above that matches.

Theorem

[Proof omitted.]

In DFS of an *un*directed graph, we get only tree and back edges. No forward or cross edges.

Topological Order

Directed acyclic graph (dag)

A directed graph with no cycles.

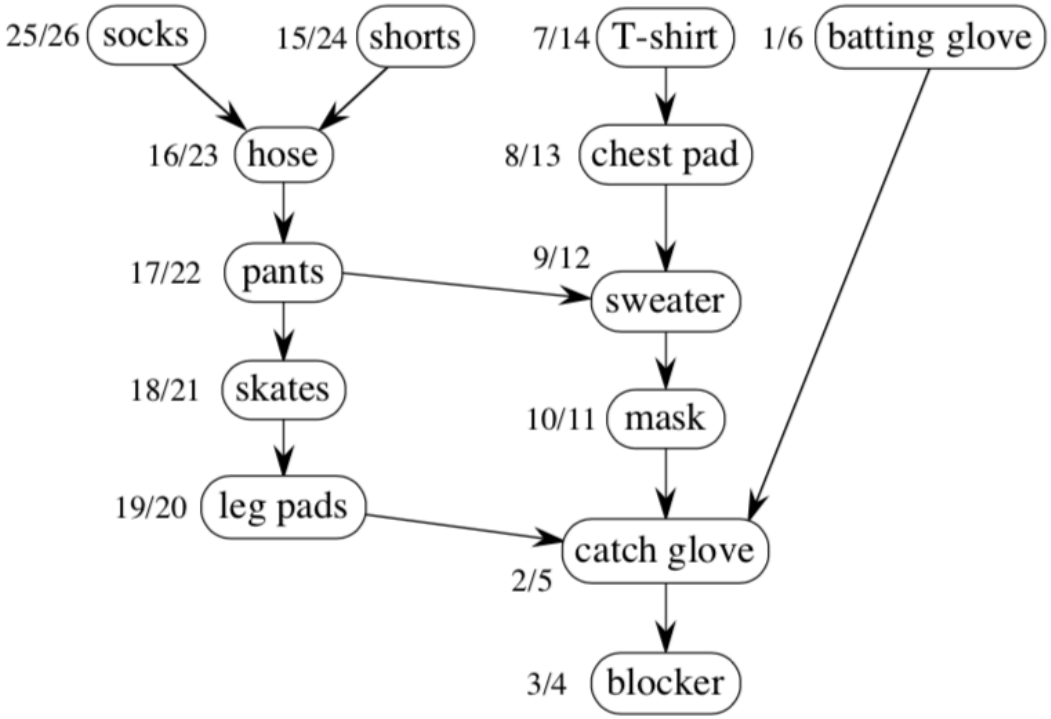
Good for modeling processes and structures that have a ***partial order:***

- $a > b$ and $b > c \Rightarrow a > c$.
- But may have a and b such that neither $a > b$ nor $b > c$.

Can always make a ***total order*** (either $a > b$ or $b > a$ for all $a \neq b$) from a partial order. In fact, that's what a topological sort will do.

Example

Example: dag of dependencies for putting on goalie equipment:



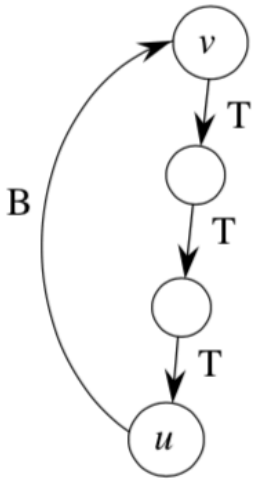
Acyclic

Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges.

Proof $\Rightarrow$: Show that back edge $\Rightarrow$ cycle.

Suppose there is a back edge (u, v). Then v is ancestor of u in depth-first forest.



Therefore, there is a path $v \rightsquigarrow u$, so $v \rightsquigarrow u \rightarrow v$ is a cycle.

$\Leftarrow$: Show that cycle $\Rightarrow$ back edge.

Suppose G contains cycle c. Let v be the first vertex discovered in c, and let (u, v) be the preceding edge in c. At time $d[v]$, vertices of c form a white path $v \rightsquigarrow u$ (since v is the first vertex discovered in c). By white-path theorem, u is descendant of v in depth-first forest. Therefore, (u, v) is a back edge. ■ (lemma)

Topological Sort

Topological sort of a dag: a linear ordering of vertices such that if $(u, v) \in E$, then u appears somewhere before v. (Not like sorting numbers.)

TOPOLOGICAL-SORT(V, E)
call DFS(V, E) to compute finishing times $f[v]$ for all $v \in V$
output vertices in order of *decreasing* finish times

Don't need to sort by finish times.

- Can just output vertices as they're finished and understand that we want the *reverse* of this list.
- Or put them onto the *front* of a linked list as they're finished. When done, the list contains vertices in topologically sorted order.

Time: $\Theta(V + E)$.

Example

Do example.

Order:

26	socks
24	shorts
23	hose
22	pants
21	skates
20	leg pads
14	t-shirt
13	chest pad
12	sweater
11	mask
6	batting glove
5	catch glove
4	blocker

Correctness

Correctness: Just need to show if $(u, v) \in E$, then $f[v] < f[u]$.
When we explore (u, v), what are the colors of u and v?

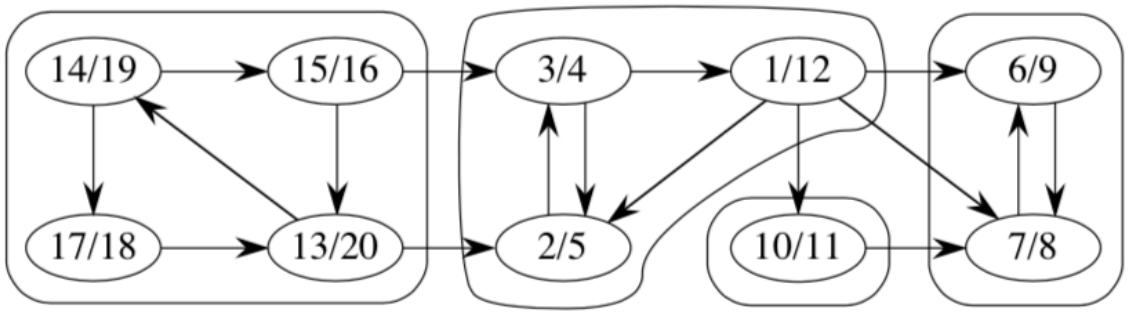
- u is gray.
- Is v gray, too?
  - *No*, because then v would be ancestor of u.
    $\Rightarrow (u, v)$ is a back edge.
    $\Rightarrow$ contradiction of previous lemma (dag has no back edges).
- Is v white?
  - Then becomes descendant of u.
    By parenthesis theorem, $d[u] < d[v] < f[v] < f[u]$.
- Is v black?
  - Then v is already finished.
    Since we're exploring (u, v), we have not yet finished u.
    Therefore, $f[v] < f[u]$.

Strongly Connected Component

Given directed graph $G = (V, E)$.

A ***strongly connected component*** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example:



Algorithm uses $G^{\mathrm{T}} =$ ***transpose*** of G.

- $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$.
- G^{T} is G with all edges reversed.

Can create G^{T} in $\Theta(V + E)$ time if using adjacency lists.

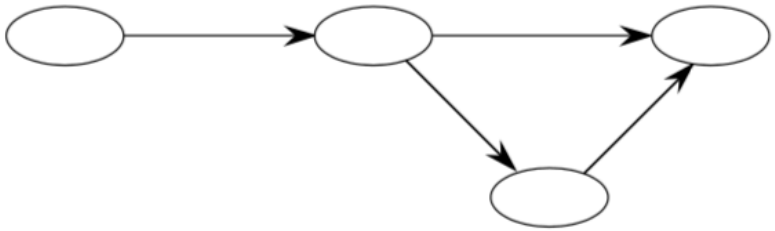
Observation: G and G^{T} have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^{T}.)

Component Graph

Component graph

- $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.

For our example:



Lemma

G^{SCC} is a dag. More formally, let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof Suppose there is a path $v' \rightsquigarrow v$ in G. Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G. Therefore, u and v' are reachable from each other, so they are not in separate SCC's. ■ (lemma)

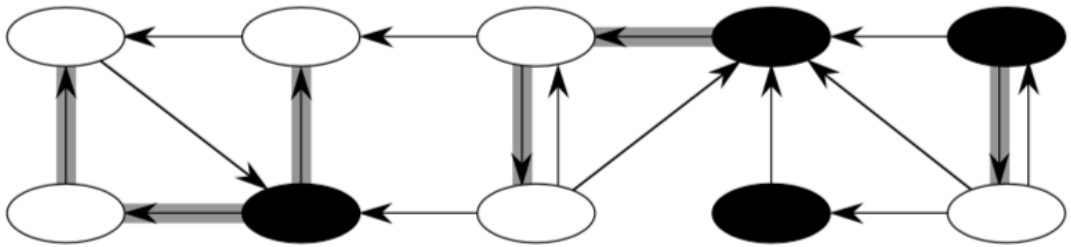
SCC(G)

call DFS(G) to compute finishing times $f[u]$ for all u
compute G^T
call DFS(G^T), but in the main loop, consider vertices in order of decreasing $f[u]$
    (as computed in first DFS)
output the vertices in each tree of the depth-first forest formed in second DFS
    as a separate SCC

Example

Example:

1. Do DFS
2. G^{T}
3. DFS (roots blackened)



Time: $\Theta(V + E)$.

How does it work

How can this possibly work?

Idea: By considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of the component graph in topological sort order.

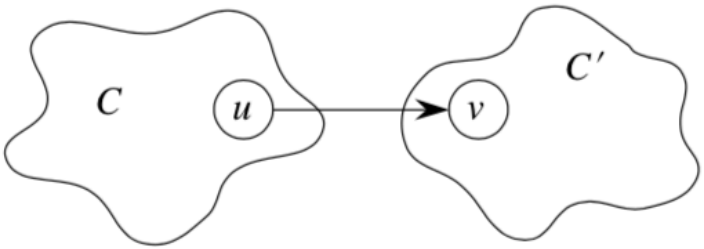
To prove that it works, first deal with 2 notational issues:

- Will be discussing $d[u]$ and $f[u]$. These always refer to *first* DFS.
- Extend notation for d and f to sets of vertices $U \subseteq V$:
  - $d(U) = \min_{u \in U} \{d[u]\}$ (earliest discovery time)
  - $f(U) = \max_{u \in U} \{f[u]\}$ (latest finishing time)

Property

Lemma

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$.



Then $f(C) > f(C')$.

Proof Two cases, depending on which SCC had the first discovered vertex during the first DFS.

Proof

- If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $d[x]$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.

  By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.

  By the parenthesis theorem, $f[x] = f(C) > f(C')$.
- If $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $d[y]$, all vertices in C' are white and there is a white path from y to each vertex in $C' \Rightarrow$ all vertices in C' become descendants of y. Again, $f[y] = f(C')$.

  At time $d[y]$, all vertices in C are white.

  By earlier lemma, since there is an edge (u, v), we cannot have a path from C' to C.

  So no vertex in C is reachable from y.

  Therefore, at time $f[y]$, all vertices in C are still white.

  Therefore, for all $w \in C$, $f[w] > f[y]$, which implies that $f(C) > f(C')$.

  ■ (lemma)

Properties

Corollary

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^{\mathrm{T}}$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof $(u, v) \in E^{\mathrm{T}} \Rightarrow (v, u) \in E$. Since SCC's of G and G^{T} are the same, $f(C') > f(C)$. ■ (corollary)

Corollary

Let C and C' be distinct SCC's in $G = (V, E)$, and suppose that $f(C) > f(C')$. Then there cannot be an edge from C to C' in G^{T}.

Proof It's the contrapositive of the previous corollary. ■

Now we have the intuition to understand why the SCC procedure works.

When we do the second DFS, on G^T, start with SCC C such that $f(C)$ is maximum. The second DFS starts from some $x \in C$, and it visits all vertices in C. Corollary says that since $f(C) > f(C')$ for all $C' \neq C$, there are no edges from C to C' in G^T.

Therefore, DFS will visit *only* vertices in C.

Which means that the depth-first tree rooted at x contains *exactly* the vertices of C.

The next root chosen in the second DFS is in SCC C' such that $f(C')$ is maximum over all SCC's other than C. DFS visits all vertices in C', but the only edges out of C' go to C, *which we've already visited.*

Therefore, the only tree edges will be to vertices in C'.

We can continue the process.

Each time we choose a root for the second DFS, it can reach only

- vertices in its SCC—get tree edges to these,
- vertices in SCC's *already visited* in second DFS—get *no* tree edges to these.

We are visiting vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in reverse of topologically sorted order.

[The book has a formal proof.]