Dynamic Programming I - Fibonacci, Shortest Paths

[MIT OpenCourseware 6.006]

- Very general and powerful design technique
- Dynamic here is used in the sense of optimization (Shortest Paths, {Minimize, Maximize} something)
- Kind of exhaustive search, which usually takes exponential time, but done in a careful way:

$$DP \simeq \text{careful brute-force} \quad \text{(version 1)}$$

Fibonacci Numbers:

- $DP \simeq$ subproblems + "reuse" (version 2)

Take a problem, split into subproblems, solve those subproblems and reuse the solutions to those subproblems

- $\begin{cases} F_1 = F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$ (Recurrence of Fibonacci Numbers)

<u>Goal:</u> compute F_n

- Naive Recursive Algorithm:

```
fib(n):
    if n ≤ 2 : f = 1
    else : f = fib(n-1) + fib(n-2)
    return f
```

$\boxed{\text{Question:}}$ How much time does the naive version requires?

Recurrence: $T(n) = T(n-1) + T(n-2) + \underbrace{\Theta(1)}_{\text{for the + operations and comparisons}}$

- Assume that $T(n-1)$ is replaced by $T(n-2)$:
  $T(n) \geq 2T(n-2)$ (lower bound)
- With each iteration we are subtracting 2 from n
- $\boxed{\text{Question:}}$ How many times can we subtract 2 from n?
  ↳ Answer: $n/2$

$$T(n) \geq 2T(n-2) = \underbrace{2 \times 2 \times 2 \times \ldots \times 2}_{n/2 \text{ times}} \underbrace{\times \Theta(1)}_{\text{for the base case}} = 2^{n/2} \times \Theta(1)$$

for the base case

$\therefore T(n)$ is exponential time

$\boxed{\text{Question:}}$ How can we make bad algorithms like this good?

↳ Answer: Memoization (DP technique)

Memoized DP algorithm

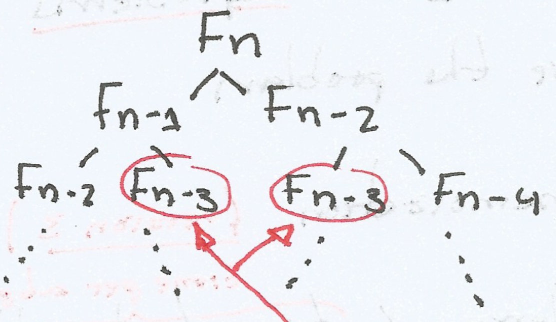
$\boxed{\text{Solve ~~subp~~ subproblem; Store subproblem solution}}$

- Idea: Whenever a Fibonacci number is computed put it in a dictionary
- memo = {} [dictionary]

```
fib(n):
    if n in memo: return memo[n]
    if n ≤ 2: f = 1
    else f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

Question: How much time does the memoized version requires?

- Helpful to think about recursion tree:



Observations:

1) Just seeing the tree we can verify the exponential growth
2) However, we are calculating the same things repeatedly in different subtrees (example)

- The first time we will have to compute Fn-3 but in the second time we will not have to compute since it will be stored in the memo table. (The same is valid for entire Fn-2 and so on...)
The cost of the second operation will be constant $\Theta(1)$

Question: Why is this efficient?

- fib(k) only recurses the first time it is called $\forall k$
- We can no longer analyze through normal recurrences
- Memoized calls cost $\Theta(1)$
- # non-memoized calls is n: fib(1), fib(2), ..., fib(n) (Number of calls)
- Non-recursive work per call is constant $\Theta(1)$
- Total time $= n \cdot \Theta(1) = \Theta(n)$

- <u>In general in DP:</u>
  - memoize (remember) "crazy term" 2 re-use solutions to <u>subproblems</u> that help solve the problem

    Big challenge in DP is determining what are the subproblems
- DP ≈ recursion + memoization <u>(version 3)</u>
- Total time: # subproblems x ~~time/subproblem~~ time per subproblem

<u>An alternative perspective:</u> Bottom-up DP algorithm

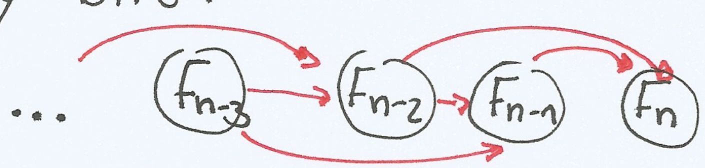
```
fib = {}   [dictionary]
for k in range(n):
    if k ≤ 2: f = 1
    else: f = fib[k-1] + fib[k-2]
    fib[k] = f
return fib[n]
```

We know start from the lower levels, i.e. in increasing order th the "bottom-up"

Total time: $\underbrace{\Theta(n)}_{\text{iterations}} \cdot \Theta(1) = \Theta(n)$

Question: Which version is more efficient? (Memoized or bottom-up)
- The last function does not have recursive call
- Instead we just have $\Theta(1)$ for the dictionary operations
- No need to maintain a function call stack
- Only requires constant space since we only need to remember the last two values

In essence both version perform a topological sort of subproblem dependency DAG:



Optimal Sub-structure

DP takes advantage of the optimal sub-structure of a problem. A problem has an optimal substructure if the optimum answer to the problem contains optimum answer to smaller subproblems.

Shortest Paths with dynamic programming

- The shortest path problem has an optimal substructure.
  Suppose $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path from u to v. This implies that $s \rightsquigarrow u$ is a shortest path from ~~s~~ s to u and this can be proven by contradiction. If there is a shorter path between s and u, we can replace $s \rightsquigarrow u$ with the shorter path in $s \rightsquigarrow u \rightsquigarrow v$ and this would yield a better path between s and v. This would contradict that $s \rightsquigarrow u \rightsquigarrow v$ was the shortest path.

- Based on this optimal substructure, we can write down the recursive formulation of the single source shortest path problem as the following:

$$\delta(s,v) = \min \{ \delta(s,u) + w(u,v) \} \quad \forall (u,v) \in E$$

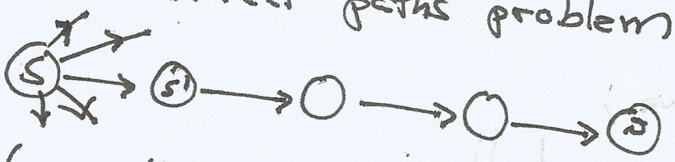
Shortest Paths:

- Compute the shortest pathway from vertex $\underline{s}$ to $\underline{v}$, for all vertex $\delta(s,v)\ \forall v$

General Idea:
- Suppose you don't know something but you would like to know the answer.
- How am I going to answer the question: guess ⇓

- Our tool will be guessing
- The algorithmic version:
  - Don't try just any guess, try them all ⇓
- DP $\simeq$ recursion + memoization + guessing (version 4)

  careful brute force
- Back to the shortest paths problem:



Idea 1: 1) Guess the first edge

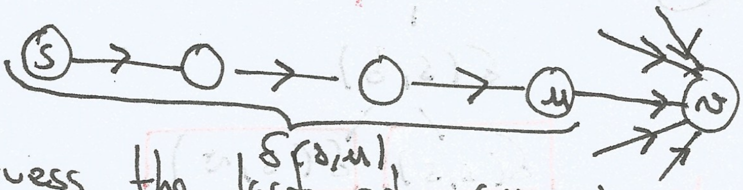
2) Recursively branch-out to the remaining nodes

Problem: The initial state is changing every time...

Although correct, this approach is difficult...

first edge approach

Idea 2:



1) Guess the last edge (u,v)

2) Recursively compute the shortest path from $\underline{s}$ to $\underline{u}$ and then add the edge guessed, i.e.:

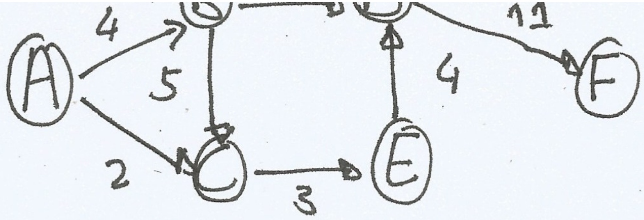
2.0) $\delta(s,s)=0$ (base case)

2.1) $\delta(s,v)=\delta(s,u)+w(u,v)$ If I am lucky and I make the right edge.

2.2) In computation / reality we are not lucky so we have to minimize:

minimizing over the choice of $\underline{u}$ since v is already ...

$$\delta(s,v)=\min_{(u,v)\in E}\Big(\delta(s,u)+w(u,v)\Big)$$



Example

$$\delta(s,v) = \min_{(u,v)\in E} \{\delta(s,u) + w(u,v)\}$$

- $\delta(A,F) = \min\{\overbrace{\delta(A,D)}^{9} + \overbrace{w(D,F)}^{11}\} = 20$ //

recursion now goes here

- $\delta(A,D) = \min\{\overbrace{\delta(A,B) + \underbrace{w(B,D)}_{10}}^{14}, \overbrace{\delta(A,E)}^{5} + \overbrace{w(E,D)}^{4}\}$

recursion now goes here

recursion now goes here

$= 9$ //

- $\delta(A,B) = \min\{\underbrace{\delta(A,A)}_{\text{base case}=0} + \underbrace{w(A,B)}_{4}\} = 0 + 4 = 4$ //

Calculated here and therefore stored in a hash

- $\delta(A,E) = \min\{\overbrace{\delta(A,C) + \underbrace{w(C,E)}_{3}}^{5}\} = 5$

- $\delta(A,C) = \min\{\overbrace{\underbrace{\delta(A,A)}_{\text{base case}=0} + \underbrace{w(A,C)}_{2}}^{2}, \delta(A,B) + \overbrace{w(B,C)}^{5}$

recursion now goes here

$= 2$ //

→ Shortest Path can be obtained if we follow the argmin of the recursion (or store something called "parent pointers")

$$\delta(A,F) = \delta(A,D) = \delta(A,E) = \delta(A,C) = \delta(A,A)$$

$\therefore A \to C \to E \to D \to F$

Question: How much time does the algorithm need?

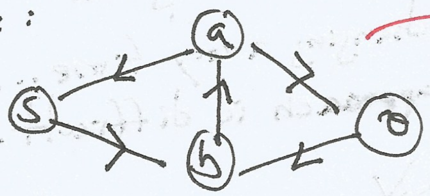
- We have a recursive algorithm without memoization...
- For every ~~guess~~ state we need to ~~perm~~ perform minimization over all other states...
- This is going to be exponential growth

Memoization version:

Check if $\delta(s,v)$ is in the table.
 If so return the value;
Otherwise:
 1) Compute value;
 2) Store in memo table;

Question: How much time does this version require?

Example:



$\delta(s,v)$
|
$\delta(s,a)$
|
$\delta(s,b)$
/ \
$\boxed{\delta(s,s)}$ $\boxed{\delta(s,v)}$

base case, recursion will stop

Notice that we have the same state that we started with, but have not yet computed...

$\therefore$ Infinite loop on graphs with cycles

Question: But what about on graphs without cycles (Direct Acyclic Graphs)?

Time = # subproblems × time/subproblem

we minimize over V subproblems

$= v \times \#incomingEdgesTo\, v$

Because this depends on v we cannot make a simple multiplication

$= \sum indeg(v) = O(E)$

In reality each subproblem takes $\Theta(\text{indegree}(v)+1)$ time, where the $\Theta(1)$ comes from a constant amount of operations. Therefore:

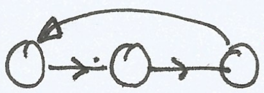
$$\text{Total time} = \sum_{v \in V} \left(\text{indegree}(v) + 1\right) = \underbrace{\Theta(E+V)}_{\checkmark}$$

$$\cancel{= \sum_{v\in V} \text{indegree}(v) + \sum_{v \in V} 1} \quad \text{By handshaking Lemma}$$

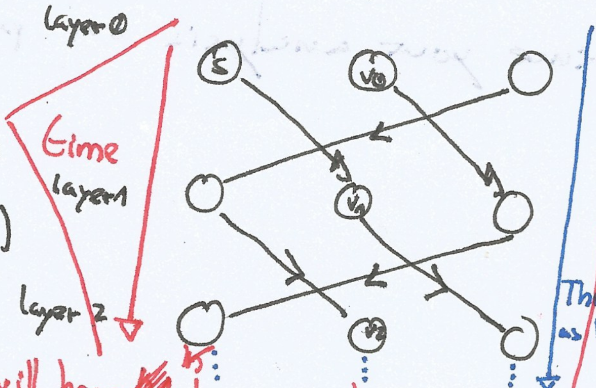
$$\cancel{= E + V = \Theta(E+V)}$$

$\therefore$ For memoization to work subproblem dependencies should be acyclic, otherwise we get an infinite-time problem

Luckily there is a technique to convert cyclic graphs into acyclic:



(cyclic grap)



The graph is now acyclic

Idea:
1) Every time I follow an edge I go down to the next layer
2) This makes every graph acyclic

We will have layers, where each layer would represent the state of the graph at a given time (look at page 8*)

$\delta_k(s,v)$ = weight of shortest $s \to v$ path that uses $\leq k$ edges

(k ↳ layer)

$$\delta_k(s, \underline{v}) = \min_{(u,v)\in E} \left(\delta_{k-1}(s,u) + w(u,v) \right)$$

We minimize for $\underline{v}$ and we have $|V|$ layers for k

$$\#\text{subproblems} = V^2$$

$$\text{Total time} : \#\text{subproblems} \times \text{time/subproblem}$$

$$= V^2 \times \Theta(\text{indegree}(V) + 1)$$

But the indegree is a function of v so we cannot express this as a multiplication

$$= V \sum_{v \in V} \text{indegree}(v) + 1$$

Question: Can we bound the number of values k?

- Each time we add a layer we are adding |V| vertices
- This will be repeated a constant number of time since our computation will not go "ad infinitum" but will eventually stop
- This means that |V| is repeated a constant amount of time, i.e.

$$k = c \cdot |V| \Rightarrow \Theta(|V|)$$

theta

- I.e. ~~at most~~ there will always be |V| layers

(now you can continue your analysis on page 7!)