CSCI203

Algorithms and Data Structures

Dynamic Programming (I)

Lecturer: Dr. Xueqiao Liu
Room: 3.117
Email: xueqiao@uow.edu.au

Dynamic Programming (DP)

- Dynamic Programming (DP) is a problem solving technique that is:
  - General;
  - Efficient;
  - Easy to understand.
- It is applicable to a wide range of different problems.
- It usually finds a solution in polynomial time…
  - … this is a GOOD THING™.
- It is often the only efficient technique we know for a problem.

Dynamic Programming

- One way to look at what it is :
  - Break the problem into sub-problems;
  - Re-use the solutions to the sub-problems.
- Solving problems defined by recurrences with overlapping subproblems
- “programming” here means “planning”

Develop a DP algorithm

- Developing steps to an optimization problem:
  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
  4. Construct an optimal solution from computed information.
- We can best see how DP works by looking at some examples.

DP #1: Fibonacci Numbers

- We are all familiar with the Fibonnaci numbers:
  - 1, 1, 2, 3, 5, 8, 13…
- Each number is defined as the sum of its two immediate predecessors:
  - $Fib_1 = Fib_2 = 1;$
  - $Fib_n = Fib_{n-1} + Fib_{n-2}$, otherwise.
- We can compute Fibonacci numbers directly from this definition.

Recursive Fibonacci

```
Procedure fib(n: integer): integer
     f: integer
     if (n<=2) then
          f = 1
     else
          f = fib(n-1) + fib(n-2)
     fi
     return f
End procedure fib
```

- This procedure is correct but it is not efficient.
- It is, in fact, an exponential time algorithm.

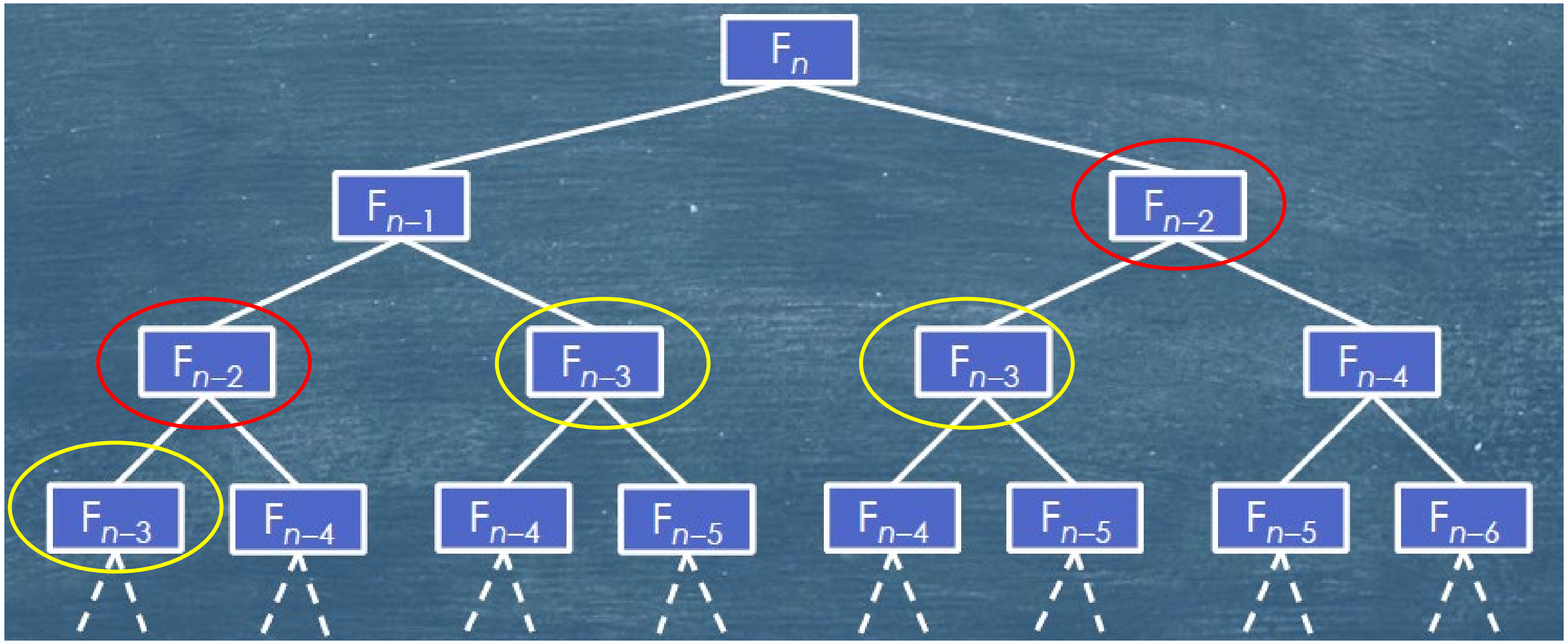
Recursive Fibonacci

- From the code we can see that the time required to compute the n^{th} Fibonacci number:
  - $T(n) = T(n-1) + T(n-2) + O(1)$
  - $T(n) > T(n-2) + T(n-2)$
  - $T(n) \in \Theta(2^{n/2})$

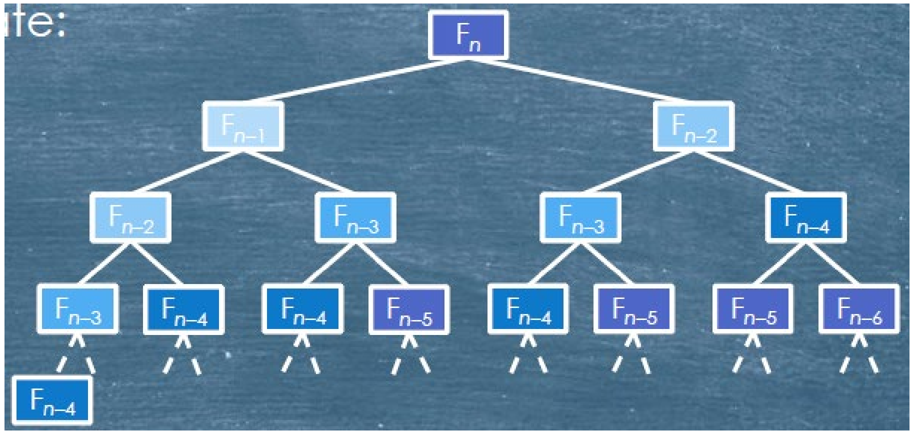
- Interestingly, the time taken to compute the $n'th$ Fibonacci number is proportional to the $n'th$ Fibonacci number.
  - e.g. 1, 1, 2, 3, 5, 8, 13…

Further Analysis

- Let us look at this another way
- Evaluating F_n requires that we evaluate the following tree:



- So, to get F_n we evaluate:
  - F_n once;
  - F_{n-1} once;
  - F_{n-2} twice;
  - F_{n-3} three times;
  - F_{n-4} five times;
  - etc.



- The cost is in the repeated evaluations of the same thing.
- What if we only evaluated each of them once?
- This is the key insight in Dynamic Programming!

Memoization: the Heart of DP

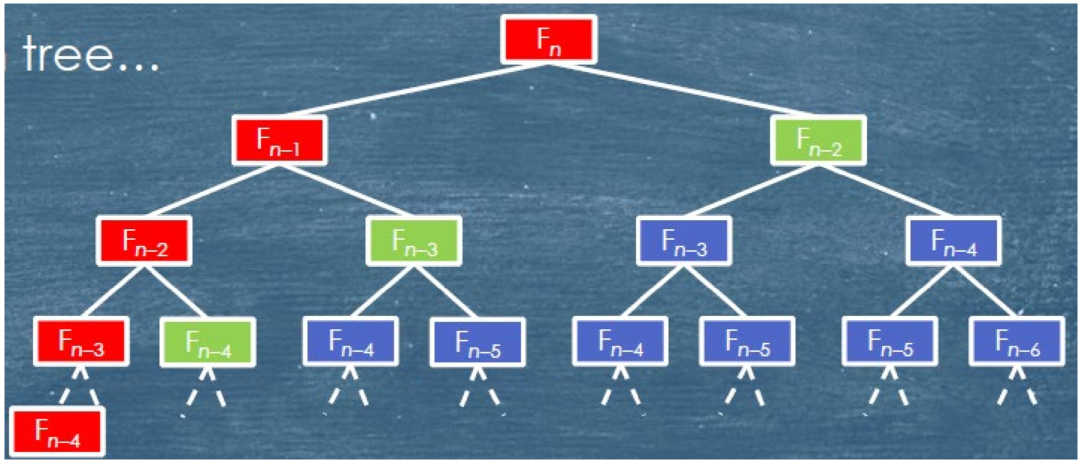
- The recognition that we only need to perform a given calculation once is central to Dynamic Programming.
- How do we remember the previous evaluations?
  - We use a dictionary;
    - A hash table.
- Let us look at the DP version of our fib procedure…

Recursive Fibonacci with Momoization

```
memo: dictionary = {}
Procedure fibDP(n: integer): integer
      f: integer
      if (n in memo) return memo[n]
      if (n<=2) then
            f = 1
      else
            f = fibDP(n-1) + fibDP(n-2)
      fi
      memo[n]=f
      return f
End procedure fibDP
```

Analysis

- Now:
  - We only recurse the first time we evaluate a given Fibonacci number.
- In all other cases we just look up the dictionary.
- Our evaluation tree…
- …becomes:
  - Evaluate;
  - Memoize;
  - Ignore.



Analysis

- With this, Dynamic Programming, approach:
  - We compute F_k once for each value $1 \le k \le n$;
    - n calls;
    - $O(1)$ per call;
  - We look up F_k once for each value $1 \le k \le n - 1$;
    - $n - 1$ calls;
    - $O(1)$ per call.
- So, fibDP takes $O(n)$ time to compute F_n

In General

- We can state the general technique for dynamic programming as follows:
  - Solve any sub-problem once and memorize (remember) these solutions for later re-use.
- In essence: DP is recursion + memoization.
- The critical problem in using DP is the identification of the sub-problems.
- The solution time for dynamic programming is derived as follows:
  - Multiply the number of distinct sub-problems by the solution time per sub-problem;

- Note: we only solve a sub-problem once.

Turning DP on its Head

- Recursion is top down solution.

- Another way to think about dynamic programming is to look at it as bottom up solution.

- We can write a bottom up Fibonacci algorithm as follows:

Bottom up Fibonacci Numbers

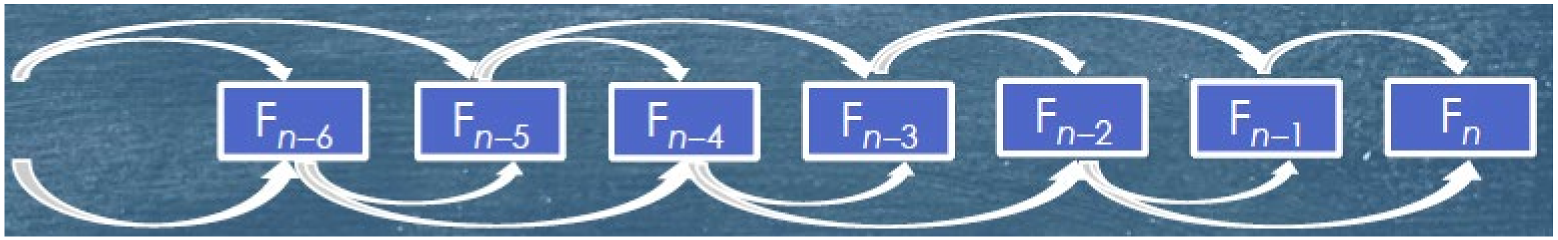
```
Procedure fibUp(n: integer): integer
        fib: dictionary = {}
        k=1
        repeat
                if k <= 2 then
                        f = 1
                else
                        f = fib[k-1]+fib[k-2]
                fi
                fib[k] = f
                k++
        until k==n
        return fib[n]
End procedure fibUp
```

1	1	…	fib[n-2]	fib[n-1]	fib[n]

- Note that this solution completely eliminates the need for recursion in calculating the nth Fibonacci number.
- *All dynamic programming algorithms can be transformed in this way.*

Bottom Up in General

- The bottom up approach to DP still involves solving the same set of sub-problems as in the top down approach.
- What changes is the order in which we solve them.
- The bottom up order can be considered as a topological sort of the problem's dependency graph.
- For the Fibonacci numbers…



- … so the sort order is $F_1, F_2, F_3, \ldots\ F_{n-3}, F_{n-2}, F_{n-1}\ F_n$.

Saving Space with DP

- Often, the bottom up version of dynamic programming allows us to save space (memory) as well as time.
- As we presented the algorithm, it used a dictionary containing n entries.
- In fact, we only ever need the last two values; we can forget the earlier ones.
- This allows us to re-write the algorithm without explicit memoization.

Memory-free Fibonacci Numbers

```
Procedure fibSmall(n: integer):integer
     prev:integer = 0
     f: integer = 1
     k=2
     repeat
          f = f+prev
          prev = f-prev
          k++
     until k == n
     return f
end procedure fibSmall
```

DP #2: Coin-row Problem

- There is a row of n coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.
- e.g.: 5, 1, 2, 10, 6, 2. What is the best selection?

DP Solution to Coin-row Problem

- Let $F(n)$ be the maximum amount that can be picked up from the row of n coins. To derive a recurrence for $F(n)$, we partition all the allowed coin selections into two groups:
  - those without last coin - the max amount is ?
    - $F(n-1)$
  - those with the last coin -- the max amount is ?
    - $c_n + F(n-2)$
- Thus we have the following recurrence
  - $F(n) \;=\; \max\{c_n + F(n-2),\; F(n-1)\}$ for $n > 1$,
  - $F(0) \;=\; 0,\; F(1) = c_1$

Algorithm

```
CoinRow(C[1..n])

//Applies formula bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array C[1..n] of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up

F[0]←0; F[1]←C[1]
for i ←2 to n do
      F[i]←max(C[i]+ F[i − 2], F[i − 1])
return F[n]
```

- Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2.

$F[0] = 0$, $F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Analysis

- Complexity
  - Space - $\Theta(n)$
  - Time - $\Theta(n)$
- The algorithm only outputs the maximum total values
- We need to back-trace the computations to see which of the two possibilities, $c_n + F(n-2)$ and $F(n-1)$ produces the maxima
  - e.g. in the last step, it was $c_6 + F(4)$ which means coin $c_6 = 2$ is part of the optimal solution
- An extra array can be used to record the information

Algorithm…

```
CoinRow(C[1..n])
//Applies formula bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array C[1..n] of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up

F[0]←0; F[1]←C[1]; S[1..n]=0
for i ←2 to n do
    //F[i]←max(C[i]+ F[i − 2], F[i − 1])
    if (C[i] + F(i-2]) > F[i-1]) then
          S[i] = 1;
          F[i] = C[i] + F(i-2]
    else
          F[i] = F[i-1]
    fi
return F[n]
```

- Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2.

$F[0] = 0,\ F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

	1					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

	1					

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

	1		1			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

	1		1	1		

$F[5] = \max\{6 + 7, 15\} = 15$

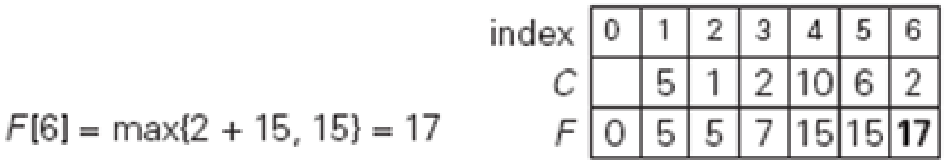
index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

	1		1	1		

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

	1		1	1		1



$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

S		1		1	1		1

$$F[1] = 5 : c_1$$
$$F[2] = 5 : c_1$$
$$F[3] = 7 : \mathrm{c}_3, \mathrm{c}_1$$
$$F[4] = 15 : c_4, c_1$$
$$F[5] = 15 : c_5, c_1$$
$$F[6] = 17 : c_6, c_4, c_1$$

DP #3: Change-making problem

- Give change for amount n using the minimum number of coins of denominations $d_1 < d_2 < \ldots < d_m$.

- We consider a dynamic programming algorithm for the general case, assuming availability of unlimited quantities of coins for each of the m denominations $d_1 < d_2 < \ldots < d_m$ where $d_1 = 1$.

DP #3: Change-making problem

- Let $F(n)$ be the minimum number of coins whose values add up to n; $F(0) = 0$.
- The amount n can only be obtained by adding one coin of denomination d_j to the amount $n - d_j$ for $j = 1, 2, \ldots, m$ such that $n \geq d_j$.
- Therefore, we can consider all such denominations and select the one minimizing
  - $F(n - d_j) + 1.$
- Since 1 is a constant, we can, of course, find the smallest $F(n - dj)$ first and then add 1 to it. Hence, we have the following recurrence for $F(n)$:

$$F(n) = \min_{j:n\geq d_j} \{F(n - d_j)\} + 1 \quad for\, n > 0$$
$$F(0) = 0$$

- We can compute $F(n)$ by filling a one-row table left to right in the manner similar to the way it was done above for the coin-row problem, but computing a table entry here requires finding the minimum of up to m numbers.

DP #3: Change-making problem

- Change-making to amount $n = 6$ and with denominations 1, 3 and 4

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

Algorithm

```
ChangeMaking(D[1..m], n)
//Applies dynamic programming to find the minimum number of coins
//of denominations $d_1 < d_2 < \ldots < d_m$ where $d_1 = 1$ that add up to a
//given amount n
//Input: Positive integer $n$ and array $D[1..m]$ of increasing positive
// integers indicating the coin denominations where $D[1] = 1$
//Output: The minimum number of coins that add up to n
F[0]←0
for i ←1 to n do
      temp←∞; j ←1
      while j ≤ m and i ≥ D[j ] do
            temp ←min(F [i − D[j ]], temp)
            j ←j + 1
      F[i]←temp + 1
return F[n]
```

Analysis

- Complexity
  - Space - $\Theta(n)$
  - Time - $\Theta(nm)$
- To find the coins of an optimal solution, we need to backtrace the computations to see which of the denominations produced the minima.
  - For the instance considered, the last application of the formula (for n = 6), the minimum was produced by d2 = 3. The second minimum (for n = 6 – 3) was also produced for a coin of that denomination. Thus, the minimum-coin set for n = 6 is two 3's.

DP #3: Change-making problem

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], \boxed{F[3-3]}\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], \boxed{F[6-3]}, F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

DP #4: Shortest Paths

- Let us apply the insights we have gained on dynamic programming to a second problem.:
  - Single source, single destination shortest path.
- We will proceed as follows:
  - Create a top down, recursive, naïve algorithm;
  - Memorize it;
  - Reconstruct it as a bottom up algorithm.
- This is a useful general approach to algorithm design in dynamic programming.

Step 1: the Naïve, Recursive Algorithm

- In deriving the naïve algorithm we need to introduce another key component of dynamic programming…
  - …guessing!
- Don't know the answer?
  - Guess!
- Don't just try any guess…
  - …try them all!
- So, DP = recursion + memoization + guessing.
- The best guess is the answer we are looking for

Some Notation for Shortest Paths

- Remember from last week:
  - Given a graph, $G = (V, E, W)$, find the shortest path from a starting vertex, $s \in V$, to all other vertices, $v \in V$;
  - $w(u, v)$ is the weight of the edge (u, v);
  - $D(s, v)$ is the length of the shortest path between s and v.
- If some vertex, u, is on the shortest path from s to v then:
  - D(s, v) = D(s, u) + D(u, v).
- Specifically, if vertex u immediately precedes vertex v in the shortest path from s to v, then:
  - $D(s, v) = D(s, u) + w(u, v)$.
- Our problem is that we don't know which vertex, u, to try…
  - …so we guess—try them all and pick the best.

The Naïve Algorithm

```
Procedure short(V{}: vertex, E{}: edge, W(): weight,
                s: vertex, v: vertex)
      if v==s then
            d=0
      else
            d = ∞
            for each u where (u,v) ∈ E
                  d = min(d, short(V, E, W, s, u) + w(u,v))
            rof
      fi
      return d
End procedure short
```

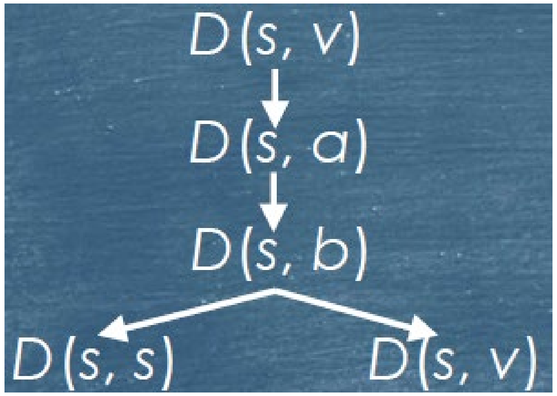
- This is a really bad algorithm:
  - We compute the shortest path between s and every other vertex repeatedly.
- It is really easy to improve, however;
- Memoize the computation.

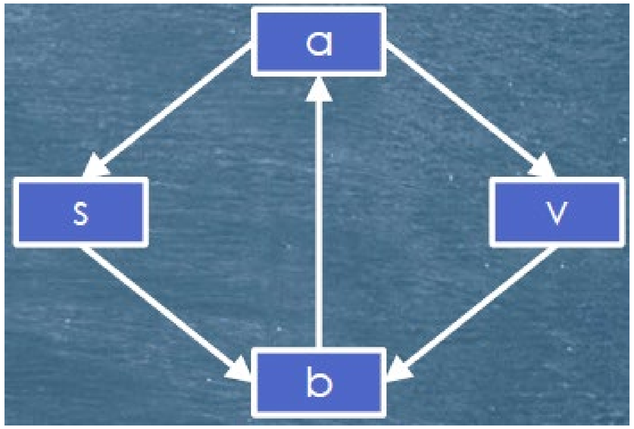
Step 2: The Memoized Algorithm

```
D: dictionary {}
Procedure shortDP(V{}: vertex, E{}: edge, W(): weight, s: vertex,
        v: vertex)
        if v==s then
                d=0
        else
                d = ∞
                for each u where (u,v) ∈ E
                  if (u in D) then
                        d=min(d, D[u]+w(u,v))
                  else
                        d = min(d, shortDP(V, E, W), s, u) + w(u,v))
                  fi
                rof
        fi
        D[v]=d
        return d
End procedure shortDP
```

Some Analysis

- Consider the following graph:
- To find the shortest path $D(s, v)$ we proceed as follows:





- We now have a problem…
  - …to find $D(s, v)$ we need to evaluate $D(s, v)$.

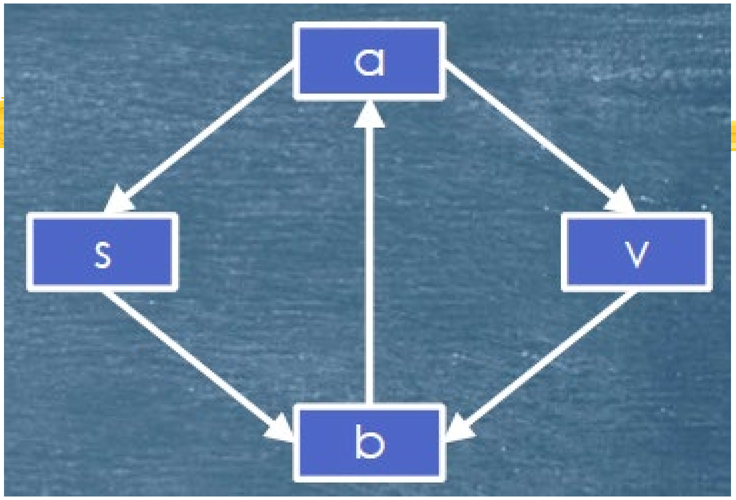
A Problem

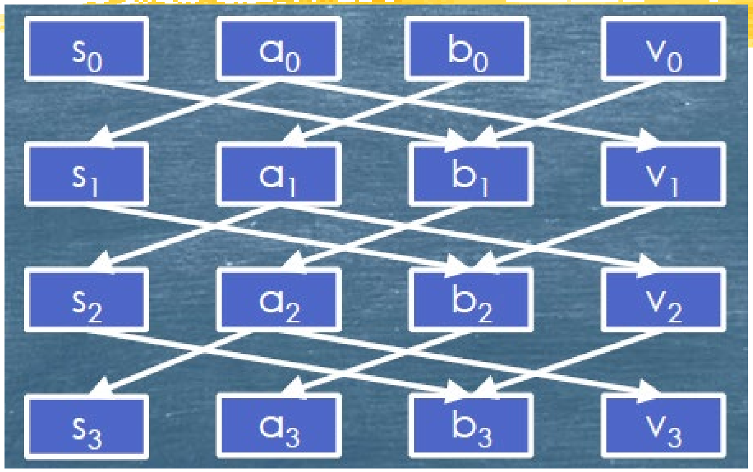
- Our "improved" algorithm has a problem.
- It takes infinite time if G has one or more cycles.
- If g is acyclic the algorithm is $O(|V| + |E|)$
- We should have anticipated this…
  - …remember the bottom up formulation.
- The order of evaluation of sub-problems is a topological sort of the dependency graph.
- You can only perform a topological sort on a DAG…
  - …no cycles allowed.

Decycling a Graph

- Is there some way to remove cycles from a graph?
- Yes, provided none of them are negative cost cycles.
- We replicate the graph $|V|$ times and construct a new graph as follows:
  - Eliminate all edges between vertices in the same copy:
  - If $(u, v) \in E$ in the original graph connect u_i to v_{i+1} in the new graph.
  - This is best seen with an example.

- Let us use our previous graph:
- This becomes:





- This new graph has $|V|^2$ vertices and $|V| \times |E|$ edges…
  - …but it has no cycles.
- We now define $D_k(s, v)$ as the shortest path from s to v that traverses exactly k edges.
- The shortest path is now the smallest of the $D_k(s, v)$ values

Analysis

- We now observe that:
  - $D_k(s,v) \ = \min(D_{k-1}(s,u) + w(u,v))$.
- So, if we use our memoized DP shortest path solution algorithm on this graph we can solve our original problem, even though our graph has cycles.
- The bottom up version of this $O(|V \times |E|)$ algorithm is exactly the same as the Bellman-Ford algorithm we saw last week.
- In fact, this is how the Bellman-Ford algorithm was discovered.

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 8.1

- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 15