Lecture

Dynamic Programming

Wilson Rivera

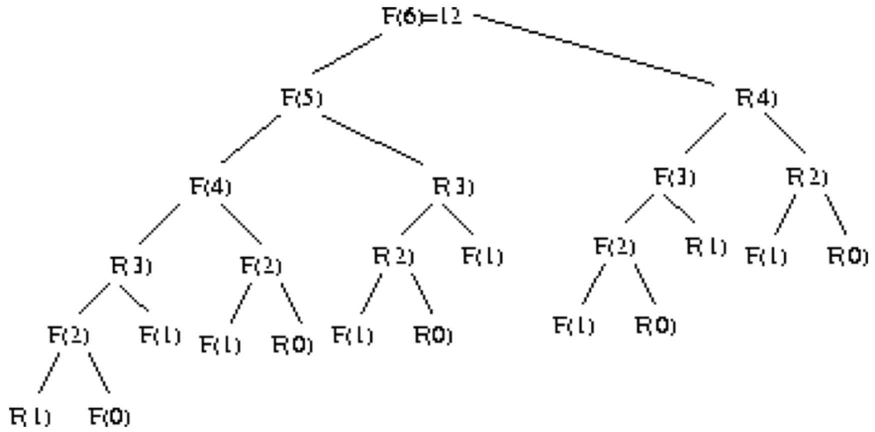
Example 1: Fibonacci number

Recursive Solution

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_2 &= F_0 + F_1 = 1 \\
F_3 &= F_1 + F_2 = 2 \\
F_4 &= F_2 + F_3 = 3 \\
F_5 &= F_3 + F_4 = 5 \\
&\vdots
\end{aligned}
$$

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Dynamic Programming Solution




```
def fib(n):
    A = [0, 1]
    for i in xrange(2, n + 1):
        A.append(A[i - 1] + A[i - 2])
    return A[n]
```

Example 2: Unique BSTs

- Given n, How many structurally unique binary search trees store vales 1 to n

```
1         3     3      2      1
 \       /     /      / \      \
  3     2     1      1   3      2
 /     /       \                 \
2     1         2                 3
```

$$A[n] = \sum_{k=0}^{n-1} A[k] * A[n-1-k]$$

```
def numTrees(n):
    A = [0 for i in range(n + 1)]
    A[0] = 1
    for i in xrange(1, n + 1):
        for k in xrange(0, i):
            A[i] += A[k] * A[i - 1 - k]
    return A[n]
```

Example 3: Edit Distance

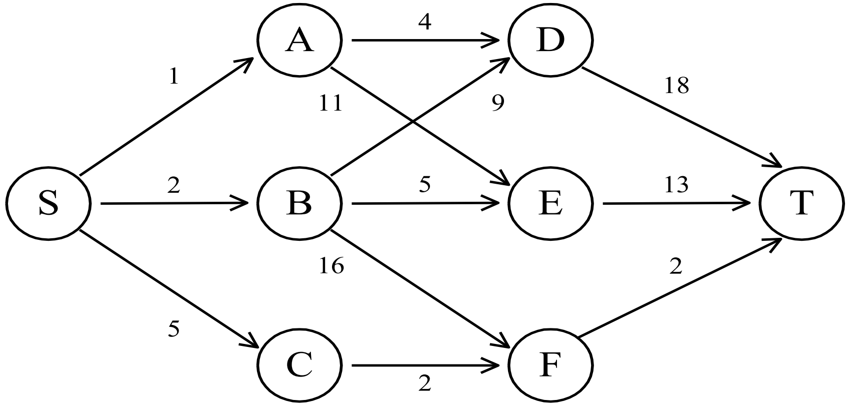
- Given two words w1 and w2, find the minimum number of steps required to convert w1 to word2. (each operation is counted as 1 step.)
- You have the following 3 operations permitted on a word
  - Insert a character
  - Delete a character
  - Replace a character

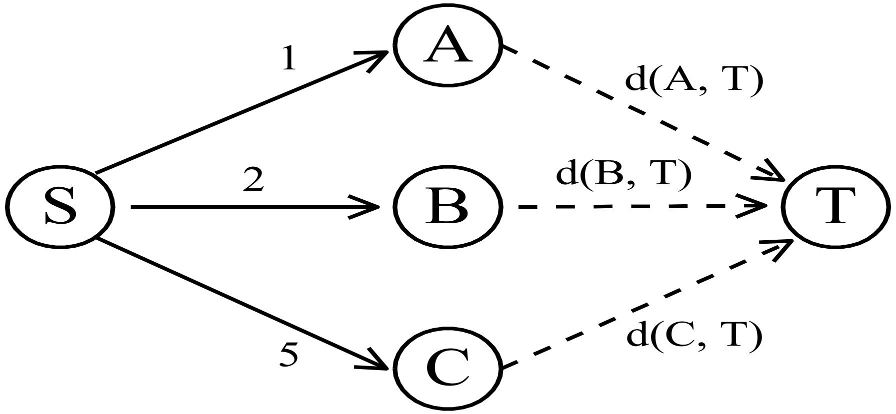
```
def minDistance(word1, word2):
    d = [[0 for j in range(len(word2) + 1)] for i in range(len(word1) + 1)]
    for i in xrange(len(word1) + 1):
        d[i][0] = i
    for j in xrange(len(word2) + 1):
        d[0][j] = j

    for i in xrange(1, len(word1) + 1):
        for j in xrange(1, len(word2) + 1):
            if word1[i - 1] == word2[j - 1]:
                d[i][j] = d[i - 1][j - 1]
            else:
                d[i][j] = 1 + min(min(d[i - 1][j], d[i][j - 1]), d[i - 1][j - 1])

    return d[len(word1)][len(word2)]
```

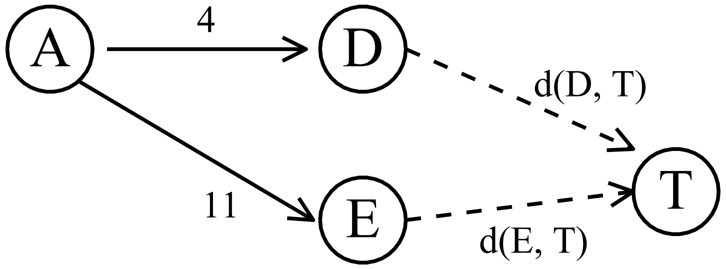
Shortest path: Dynamic programming





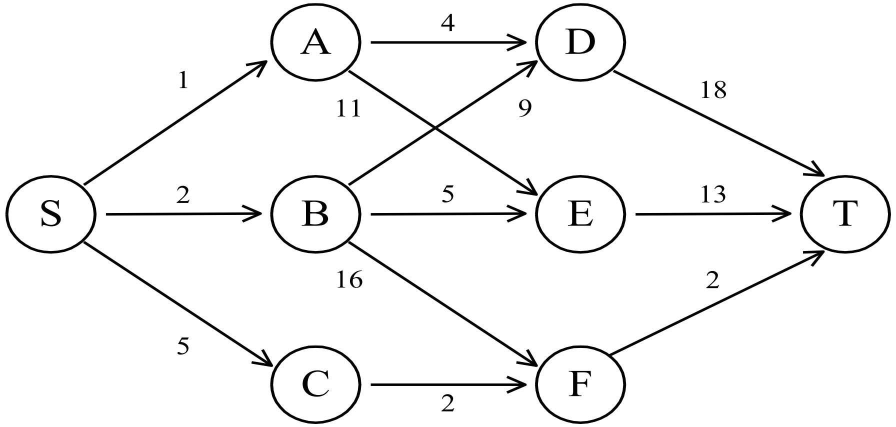
d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}

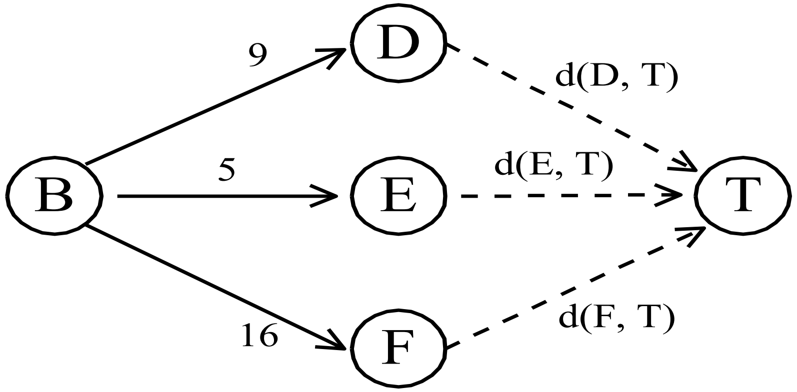
- d(A,T) = min{4+d(D,T), 11+d(E,T)}
  = min{4+18, 11+13} = 22.



- $d(B, T) = \min\{9+d(D, T), 5+d(E, T), 16+d(F, T)\}$

  $= \min\{9+18, 5+13, 16+2\} = 18.$

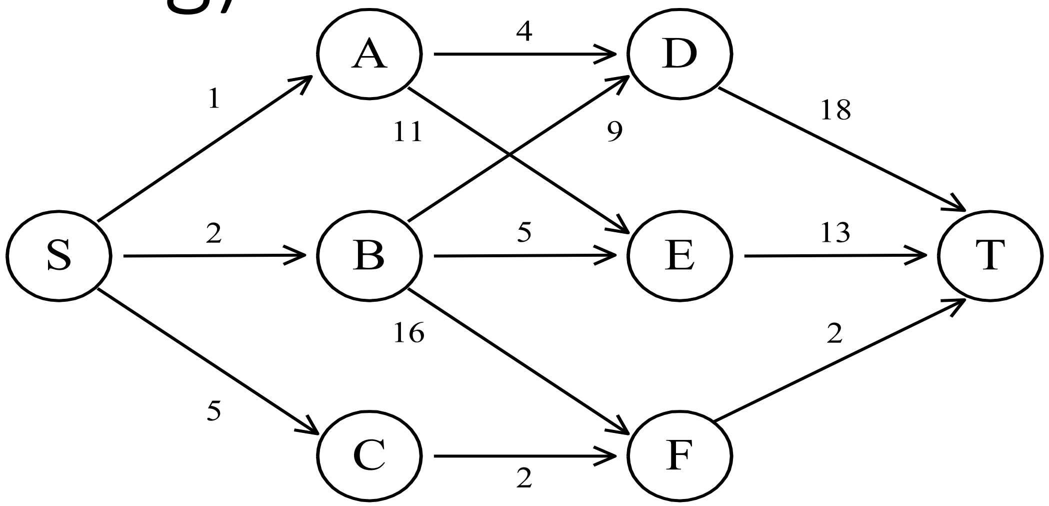




- $d(C, T) = \min\{ 2+d(F, T) \} = 2+2 = 4$
- $d(S, T) = \min\{1+d(A, T), 2+d(B, T), 5+d(C, T)\}$

  $= \min\{1+22, 2+18, 5+4\} = 9.$

Backward approach (forward reasoning)

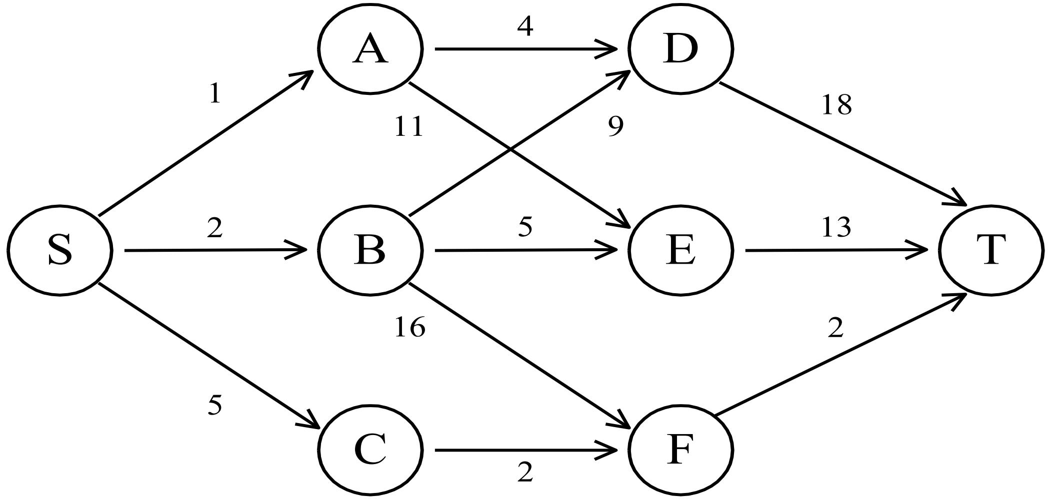


- d(S, A) = 1
  d(S, B) = 2
  d(S, C) = 5
- d(S,D)=min{d(S,A)+d(A,D), d(S,B)+d(B,D)}
  = min{ 1+4, 2+9 } = 5
  d(S,E)=min{d(S,A)+d(A,E), d(S,B)+d(B,E)}
  = min{ 1+11, 2+5 } = 7
  d(S,F)=min{d(S,B)+d(B,F), d(S,C)+d(C,F)}
  = min{ 2+16, 5+2 } = 7

- $d(S,T) = \min\{d(S, D)+d(D, T), d(S,E)+ d(E,T), d(S, F)+d(F, T)\}$

  $= \min\{ 5+18, 7+13, 7+2 \}$

  $= 9$



Principle of optimality

- Principle of optimality: Suppose that in solving a problem, we have to make a sequence of decisions $D_1, D_2, \ldots, D_n$. If this sequence is optimal, then the last k decisions, $1 < k < n$ must be optimal.
- e.g. the shortest path problem

  If $i, i_1, i_2, \ldots, j$ is a shortest path from i to j, then $i_1, i_2, \ldots, j$ must be a shortest path from i_1 to j
- In summary, if a problem can be described by a multistage graph, then it can be solved by dynamic programming.

Dynamic Programming

- **Dynamic Programming** is a basic paradigm in algorithm design used to solve problems by relying on intermediate solutions to smaller sub-problems.
  - **Optimal substructure**.
    - Optimal solutions to sub-problems are sub-solutions to the optimal solution of the original problem.
  - **Overlapping sub-problems**.
    - The sub-problems show up again and again

Dynamic Programming

- **Like divide and conquer**, DP solves problems by combining solutions to sub-problems.
- **Unlike divide and conquer**, sub-problems are not independent.
  - Sub-problems may share sub-sub-problems (overlapping)
  - However, solution to one sub-problem may not affect the solutions to other sub-problems of the same problem.
- DP **reduces computation** by
  - Solving sub-problems in a bottom-up fashion.
  - Storing solution to a sub-problem the first time it is solved.
  - Looking up the solution when sub-problem is encountered again.

Steps in Dynamic Programming

1. Identify **optimal substructure**
   - How to break up an optimal solution into optimal sub-solution with overlapping
2. Define value of **optimal solution recursively**
   - write down a recursive formulation
3. Compute optimal solution values either top-down with caching or bottom-up in a table
   - **dynamic programming algorithm**

Longest Common Subsequence (LCS)

- We say that a **sequence** Z is a subsequence of a sequence X if Z can be obtained from X by deleting symbols.
- a sequence Z is a **longest common subsequence (LCS)** of X and Y if Z is a subsequence of both X and Y , and any sequence longer than Z is not a subsequence of at least one of X or Y

Longest Common Subsequence (LCS)

$C[i, j] =$ length of LCS$(X[1:i], Y[1:j])$.

$$C[i,j] = \begin{cases} 1 + C[i-1, j-1], & \text{if } X[i] = Y[j] \\ \max(C[i-1, j], C[i, j-1]), & \text{otherwise} \end{cases}$$

Algorithm 1: lenLCS(X, Y)

Initialize an $n + 1 \times m + 1$ zero-indexed array C.
Set $C[0, j] = C[i, 0] = 0$ for all $i, j \in \{1, \ldots, m\} \times \{1, \ldots, n\}$.
for $i = 1, \ldots, m$ **do**
  for $j = 1, \ldots, n$ **do**
    if $X[i] = Y[j]$ **then**
      $C[i, j] \leftarrow C[i-1, j-1] + 1$
    else
      $C[i, j] \leftarrow \max\{C[i-1, j], C[i, j-1]\}$
return C

O(mn)

Longest Common Subsequence (LCS)

- A = b a c a d   B = a c c b a d c b

B

			a	c	c	b	a	d	c	b
		0	0	0	0	0	0	0	0	0
	b	0	0	0	0	1	1	1	1	1
	a	0	1	1	1	1	2	2	2	2
A	c	0	1	2	2	2	2	2	3	3
	a	0	1	2	2	2	3	3	3	3
	d	0	1	2	2	2	3	4	4	4

- After all $L_{i,j}$'s have been found, we can trace back to find the longest common subsequence of A and B.

Tracking Back LCS

Algorithm 2: LCS(X, Y, C)

```
// C is filled out already in Algorithm 1
L ← ∅
i ← m
j ← n
while i > 0 and j > 0 do
    if X[i] = Y[j] then
        append X[i] to the beginning of L
        i ← i − 1
        j ← j − 1
    else if C[i, j] = C[i, j − 1] then
        j ← j − 1
    else
        i ← i − 1
```

$O(n + m)$

summary

- Dynamic programming
  - Optimal substructure
  - Overlapping sub-problems
- DP problem examples
  - Fibonacci numbers
  - Unique BSTs
  - Shortest path
  - Longest common subsequences