ADVANCED ANALYSIS OF ALGORITHMS

UNIT 9: DYNAMIC PROGRAMMING. PROPERTIES AND STRATEGY. ANALYSIS. LIMITATIONS.

ALGORITHMIC PARADIGMS

- Greedy. Build up a solution incrementally, myopically optimizing some local criteria.
- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem *independently*, and combine solutions to sub-problems to form a solution to the original problem.
- Dynamic programming. Break up a problem into a series of overlapping sub-problems and build up solutions to larger and larger sub-problems. DP ~ is a "careful brute force"
- Unlike divide and conquer, sub-problems are not independent. Sub-problems may share sub-sub-problems.

❑ DEFINITION

Dynamic programming (also known as **dynamic optimization**) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions (Wikipedia).

❑ DEFINITION

It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)

Finds solutions to subproblems and stores them in memory. It combines (reuses) them somehow to find a solution to a slightly larger subproblem

More efficient than "*brute-force methods*", which solve the same subproblems over and over again

DYNAMIC PROGRAMMING HISTORY

- Richard Bellman pioneered the systematic study of DP in the 1950s.
- Etymology
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research
  - Bellman sought an impressive name to avoid confrontation
    - “It's impossible to use dynamic in a pejorative sense"
    - “Something, not even a congressman could object to“
    - DP term sounded cool ☺ !

DYNAMIC PROGRAMMING APPLICATIONS

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ….

Some famous dynamic programming algorithms.

- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

1. Real-world Use Cases of Dynamic Programming

OPTIMAL SUBSTRUCTURE PROPERTY

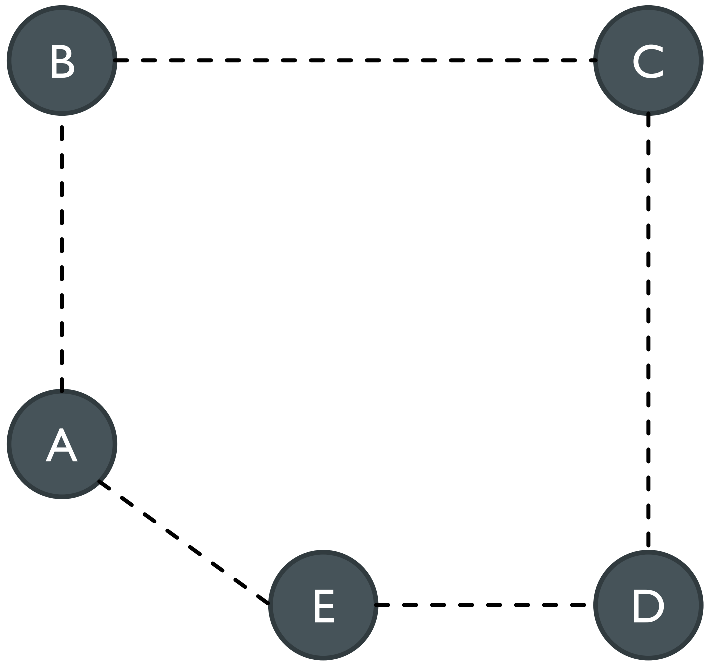
- If the optimal solution to a problem P, of size n, can be calculated by looking at the optimal solutions to subproblems $[p1, p2, \dots]$(not all the sub-problems) with size less than n, then this problem P is considered to have an optimal substructure.
- If S is an optimal solution to a problem, then the components of S are optimal solutions to subproblems
- When the problem lacks optimal substructure, a solution is to **"reconstruct"** the problem.
- How to prove that an optimal solution is composed of optimal solutions to subproblems? **"proof by contradiction"**

OPTIMAL SUBSTRUCTURE PROPERTY

- Examples:
  - True for single-source shortest path
  - True for knapsack
  - True for coin-changing
  - Not true for longest-simple-path
  - Not true for Maximum Clique Problem

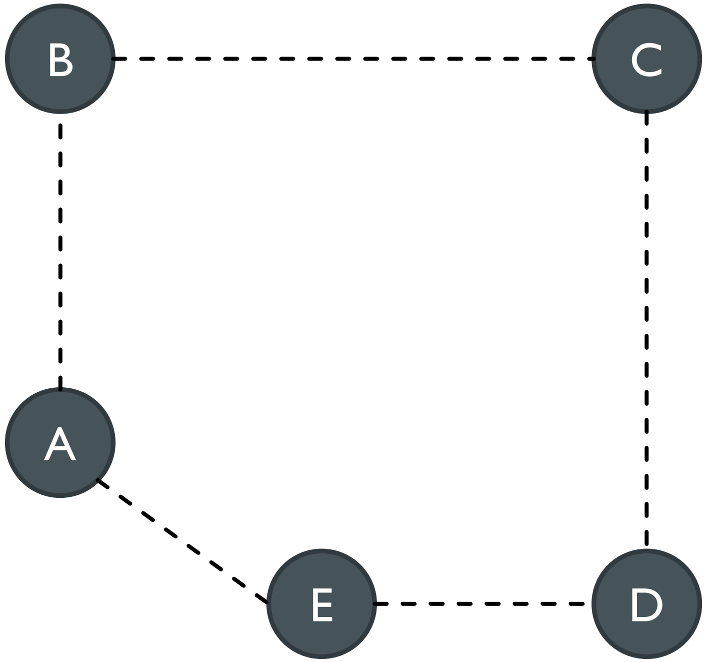
1. Optimal Substructure Property in Dynamic Programming
2. Optimal Substructure Property

❑ LONGEST PATH PROBLEM



- **Goal:** Find longest path between two vertices without repeating an edge.
- Longest (A, C): A → E → D → C
- ***If*** the principle of optimality applies to Longest Path Problem: Then we should be able to split the Problem into Sub Parts

❑ LONGEST PATH PROBLEM



- Longest (A, C): A → E → D → C
  can be done by Longest (A, D) +(D, C)?

Longest (A, D) = A→B→C→D

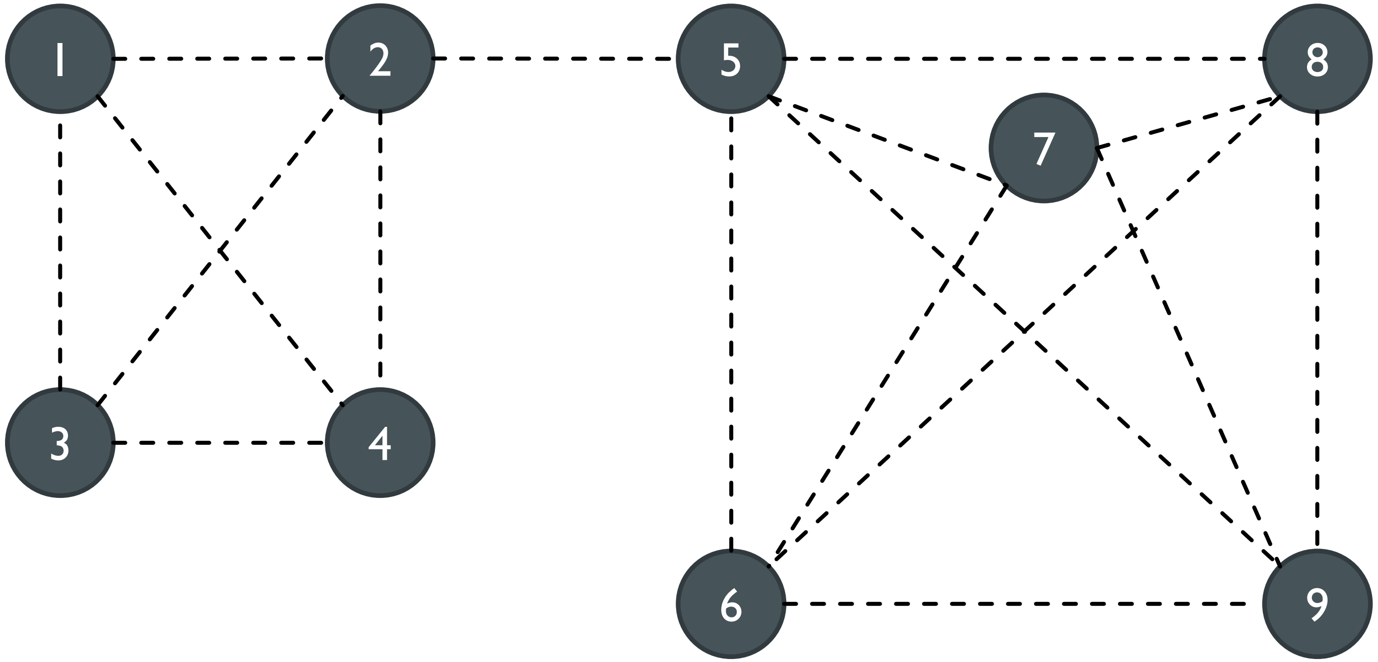
Longest(A,D)+ (D,C)
(C, D) and (D, C) is same edge!!

→ The sub-solutions do not combine to form the overall optimal solution.

→ The Longest Path Problem does not exhibit the Optimal Structure

→ Not a candidate problem for a Dynamic Programming Solution

MAXIMUM CLIQUE PROBLEM

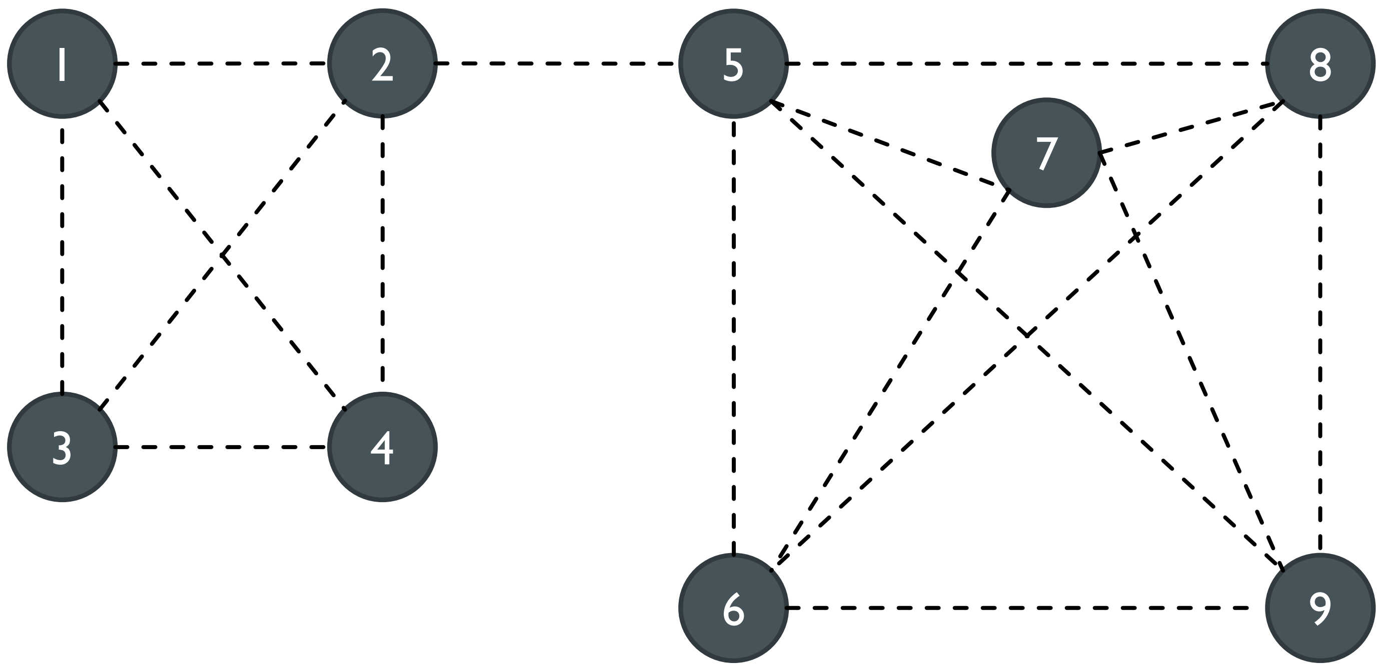


- Definition: Clique – vertices are all attached to each other.

- {1, 2, 3, 4} = clique
- {5, 6, 7, 8, 9} = clique

- Definition: Maximal Clique – A clique with the most vertices in a graph
- Vertices = {1, 2, 3, 4, 5, 6, 7, 8, 9}; Maximal clique = {5, 6, 7, 8, 9}

❑ MAXIMUM CLIQUE PROBLEM



- If we split the graph into Vertices = {1, 2, 3, 4, 5, 6, 7} + {8,9} will we obtain the same maximal clique?
- Vertices = {1, 2, 3, 4, 5, 6, 7};
  Maximal clique = {1, 2, 3, 4}
- Maximal clique != {5, 6, 7, 8, 9}

- Ca cannot break down the set of vertices into smaller sub problems and maintain the overall optimal solution
- This problem does not exhibit the optimal sub structure.
- This problem is not candidate for a dynamic programming solution.

STEPS IN DYNAMIC PROGRAMMING

Characterize	Characterize the structure of an optimal solution.
Define	Define value of optimal solution recursively.
Compute	Compute optimal solution values either top-down with caching or bottom-up in a table.
Construct	Construct an optimal solution from computed values.

Problem:

Let's consider the calculation of **Fibonacci** numbers:

$$F(n) = F(n-2) + F(n-1)$$

with seed values $F(1) = 1, F(2) = 1$ or $F(0) = 0, F(1) = 1$

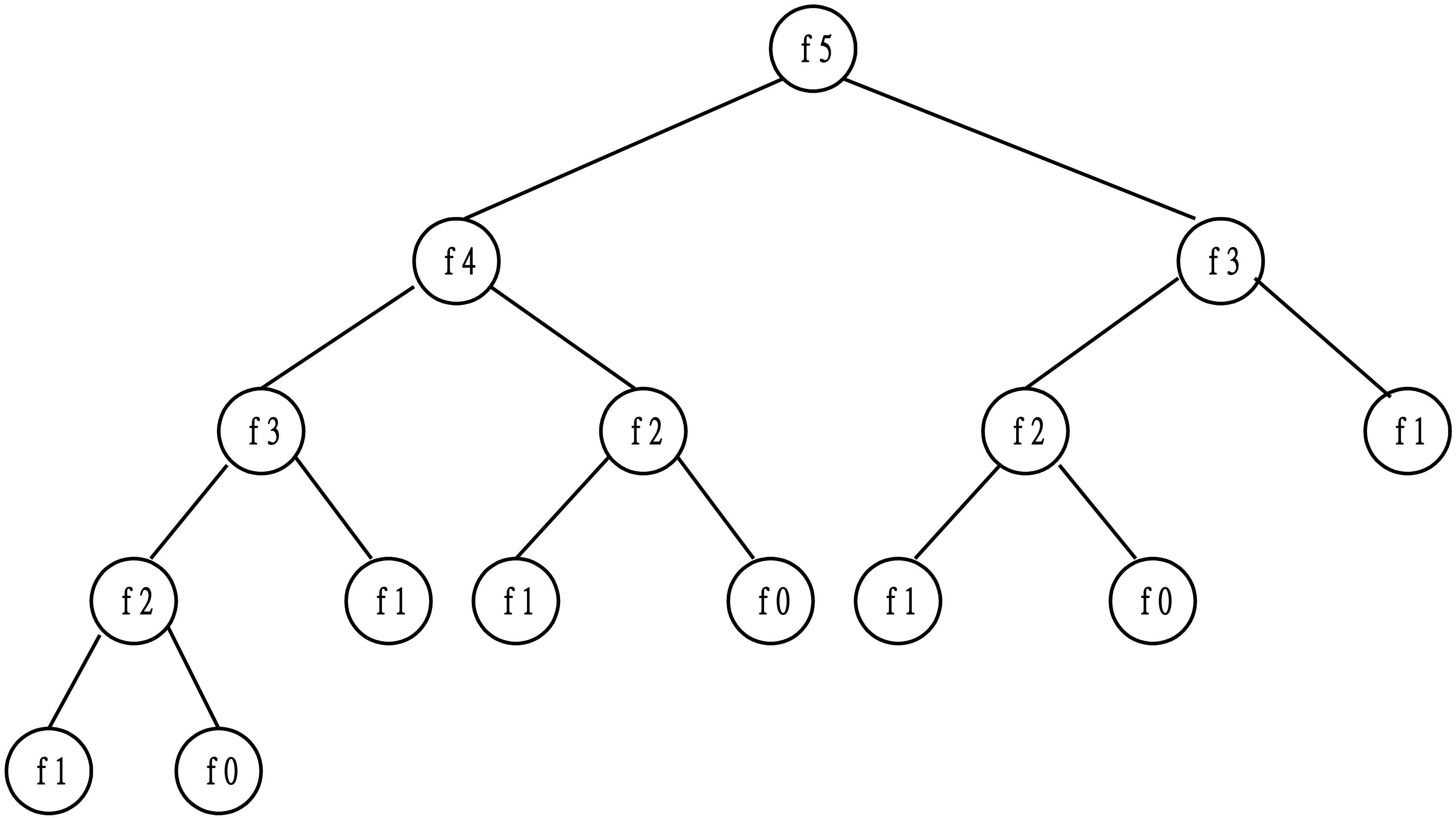
What would a series look like:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Naïve Recursive Algorithm:

```
Fib(n){

    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

   Return Fib(n-1)+Fib(n-2)
}
```

DYNAMIC PROGRAMMING



- Time Complexity

for $n > 1$: $T(n) = T(n-1) + T(n-2) + 4$
(1 comparison, 2 subtractions, 1 addition)

```
int fib(int n) {
    if (n <= 1)return n;
    return fib(n - 1) + fib(n - 2);
}
```

Let's say $c = 4$ and try to first establish a lower bound by approximating that $T(n-1) \sim T(n-2)$, though $T(n-1) \geq T(n-2)$, hence lower bound

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &= 2.T(n-2) + c \quad //from\ the\ approximation\ T(n-1) \sim T(n-2) \\ &= 2.(2T(n-4) + c) + c \\ &= 4.T(n-4) + 3c \\ &= 8.T(n-6) + 7c \\ &= 2^k.T(n - 2k) + (2^k - 1) * c \end{aligned}$$

Let's find the value of k for which: $(n - 2k = 0)$➔$k = n/2$

$$\begin{aligned} T(n) &= 2^{(n/2)} * T(0) + (2^{(n/2)} - 1) * c \\ &= 2^{(n/2)} * (1 + c) - c \end{aligned}$$

$i.e., T(n) \sim 2^{(n/2)}$

- Time Complexity

```
int fib(int n) {
    if (n <= 1)return n;
    return fib(n - 1) + fib(n - 2);
}
```

For the upper bound we can approximate $T(n-2) \sim T(n-1)\ as\ T(n-2)\ \leq\ T(n-1)$

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&= 2.T(n-1) + c \quad //from\ the\ approximation\ T(n-1) \sim T(n-2) \\
&= 2.(2.T(n-2) + c) + c \\
&= 4.T(n-2) + 3c \\
&= 8.T(n-3) + 7c \\
&= 2^k.T(n-k) + (2^k - 1) * c
\end{aligned}
$$

Let's find the value of k for which: $(n - k = 0)$ ➔ $k = n$

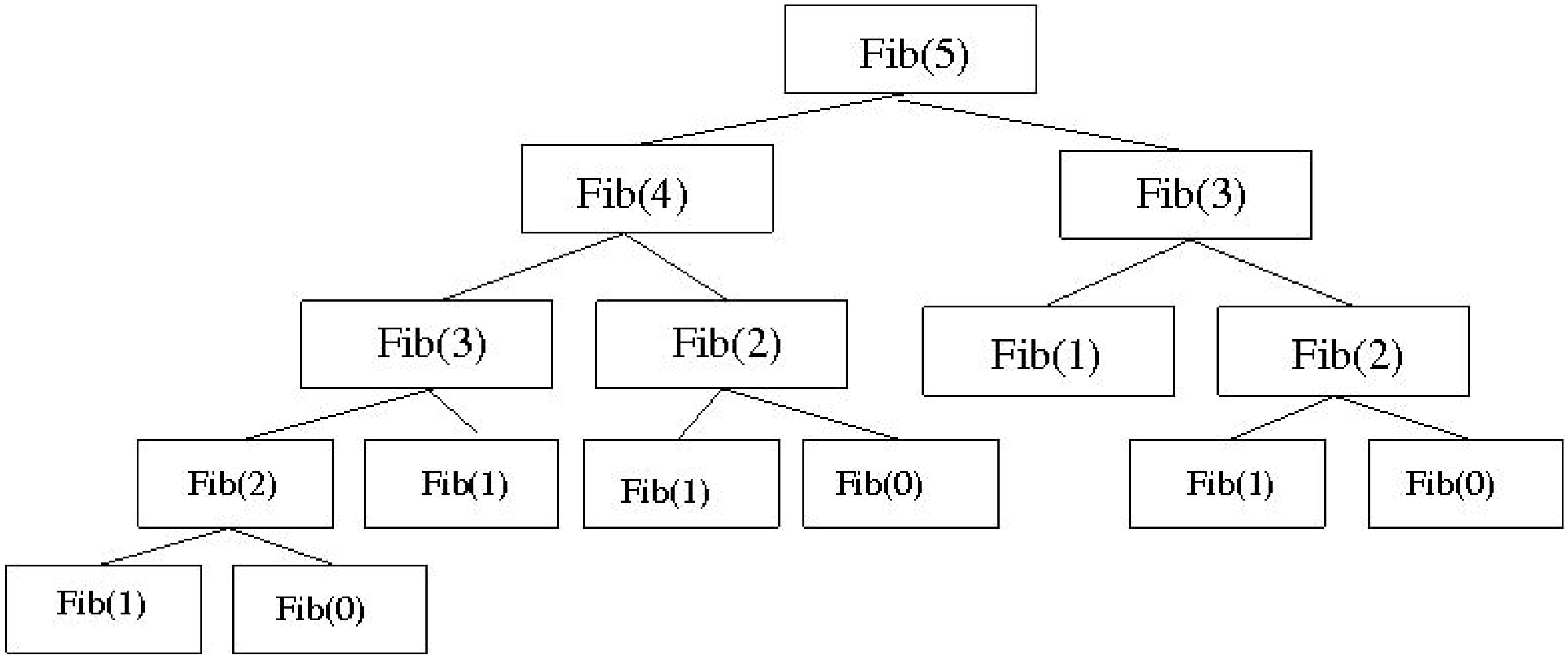
$$
\begin{aligned}
T(n) &= 2^n * T(0) + (2^n - 1) * c \\
&= 2^n * (1 + c) - c
\end{aligned}
$$

$i.e., T(n) \in O(2^n)$

Hence the time taken by recursive Fibonacci $\in O(2^n)$or exponential. Space memory of $O(n)$

Recursion tree

What's the problem?



MEMOIZATION (A TECHNIQUE OF DP)

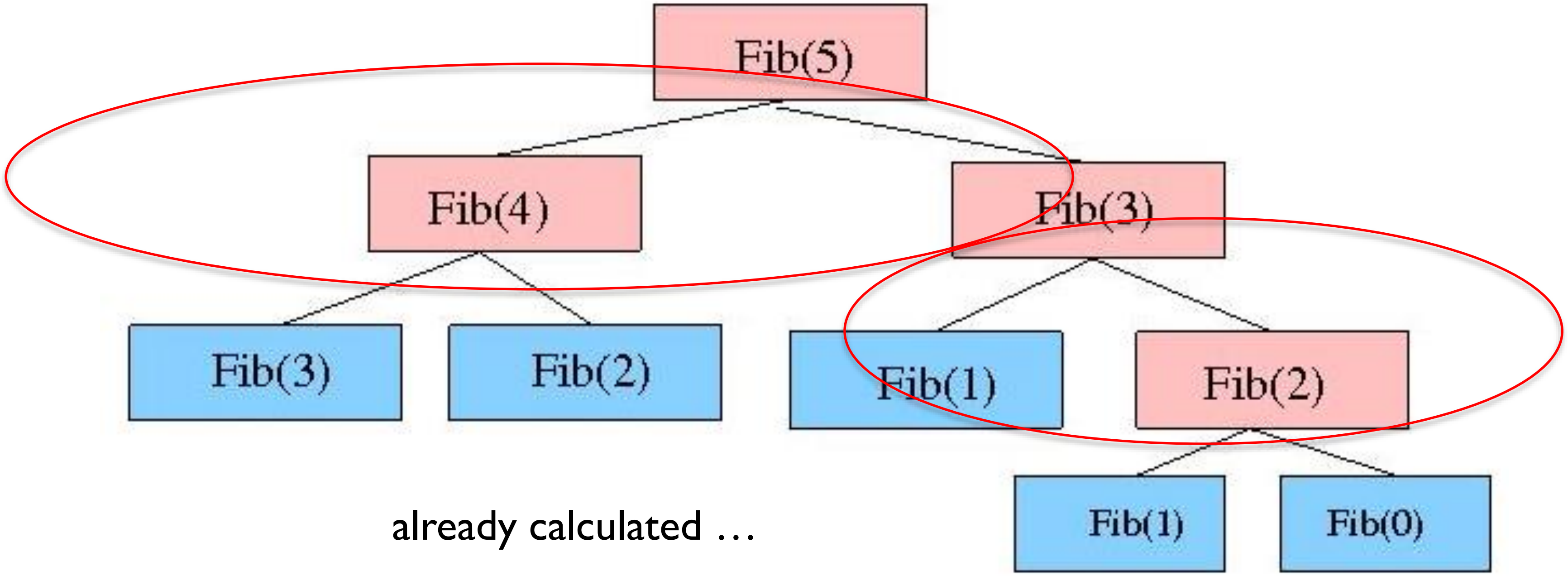
- Another technique: **Memoization** *(Memo means remember)*
  - AKA using a *memory function*
  - General procedure: It works for any recursive algorithm
- Simple idea:
  - Calculate and store solutions to subproblems
  - Before solving it (again), look to see if you've remembered it
  - ➔ Recursion + Memory

MEMOIZATION

- Use a Table abstract data type
  - Lookup key: whatever identifies a subproblem
  - Value stored: the solution
- Could be an array/vector
  - $E.g.$, for Fibonacci, store $\boldsymbol{fib(n)}$ using index $\boldsymbol{n}$
  - Need to initialize the array
- Could use a map / hash-table

```
Fib(n){
    if (n == 0) return memo[0];
    if (n == 1)  return memo[1];

    if (Fib(n-2)) is not already calculated)
        call Fib(n-2);

    if(Fib(n-1)) is not already calculated)
        call Fib(n-1);

    //Store the n^th Fibonacci no. in memory & use previous results.
    memo[n] = memo[n-1] + memo[n-2]

    Return memo[n];
}
```

MEMOIZATION



already calculated …

DYNAMIC PROGRAMMING (TOP DOWN+ MEMO) | ANALYSIS

- ❑ Fib(k) only recurses the first time it is called, $\forall\ k$
- ❑ Memoized calls cost $\Theta(1)$
- ❑ Number of non-memorized calls (sub-problems) is n
  Fib(1), Fib(2), … , Fib(n)
- ❑ Non-recursive work per call (Time per sub-problem) == $\Theta(1)$
- ❑ ➔ Time = $\Theta(n)$

Time: $O(n)$
Space !! $O(n)$

- ❑ **Time = (Number of sub-problems $\times$ Amount of time per sub-problem) $+$ (Time to combine sub-problems)**

DYNAMIC PROGRAMMING (ANOTHER TECHNIQUE: BOTTOM UP)

- Fibonacci number is sum of previous two Fibonacci numbers $f(n) = f(n-1) + f(n-2)$
- First 10 Fibonacci numbers are $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$

```java
//Iterative version of Fibonacci number
public static long fibIter(long n) {
    if(n==0) return 0;
    if (n == 1) return 1;

    long f1 = 0, f2 = 1, fi = 1;

    for (int i = 1; i <= n; i++) {
        fi = f1 + f2;
        f1 = f2;
        f2 = fi;
    }

    return fi;
}
```

```java
int number = 10;
System.out.println("Fibonacci series upto " +
number + " numbers : ");
for (int i = 1; i < number; i++) {
    System.out.print(fibIter(i) + " ");
}
```

```
Fibonacci series up to 10 numbers :
1 1 2 3 5 8 13 21 34 55 89
```

Only does real work for values it hasn't seen before.
Bottom → Top (No recursion! Save memory space)

Linear, it runs in $O(n)$ time
Memory space is $O(1)$

DYNAMIC PROGRAMMING

f1	f2	fi									
0	1	1									
	f1	f2	fi								
0	1	1	2								
		f1	f2	fi							
0	1	1	2	3							
0	1	1	2	3	5						
0	1	1	2	3	5	8					
0	1	1	2	3	5	8	13				
0	1	1	2	3	5	8	13	21			
0	1	1	2	3	5	8	13	21	34		
0	1	1	2	3	5	8	13	21	34	55	
0	1	1	2	3	5	8	13	21	34	55	89

- Main approach:
  recursive, holds answers to a sub problem in a table, can be used without recomputing.

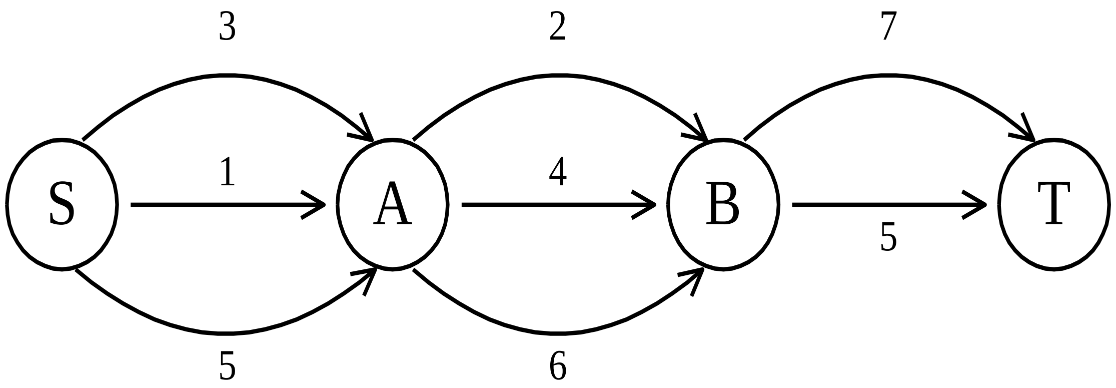
- Can be formulated both via recursion and saving results in a table (*memoization*).

- Typically, we first formulate the recursive solution and then turn it into recursion plus dynamic programming via *memoization* or bottom-up.

- To find a shortest path in a multi-stage graph
- Apply the greedy method:

the shortest path from S to T :

$$1 + 2 + 5 = 8$$



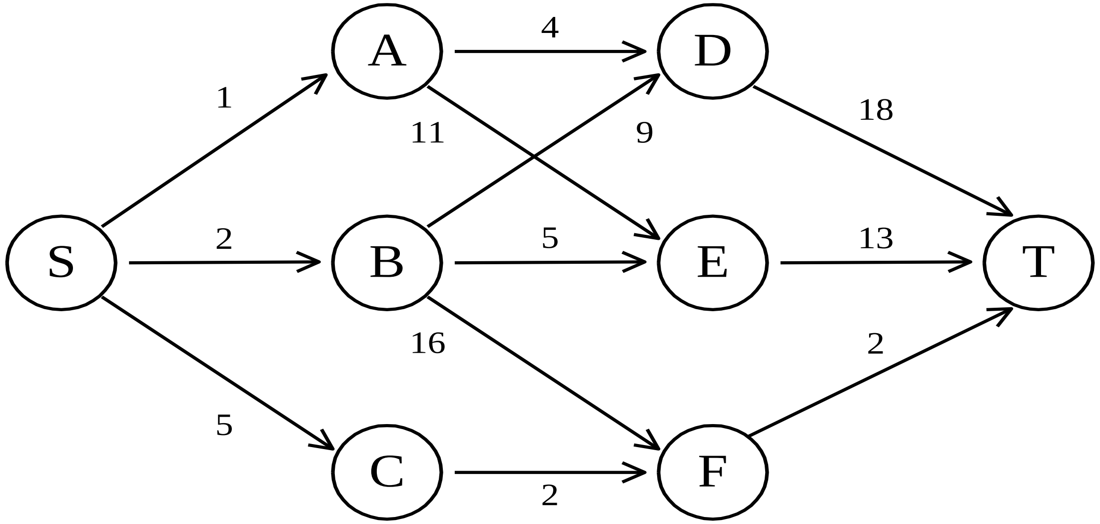
SHORTEST PATH PROBLEM

- The greedy method can not guarantee optimality to this case:

$$(S, A, D, T) = 1 + 4 + 18 = 23$$

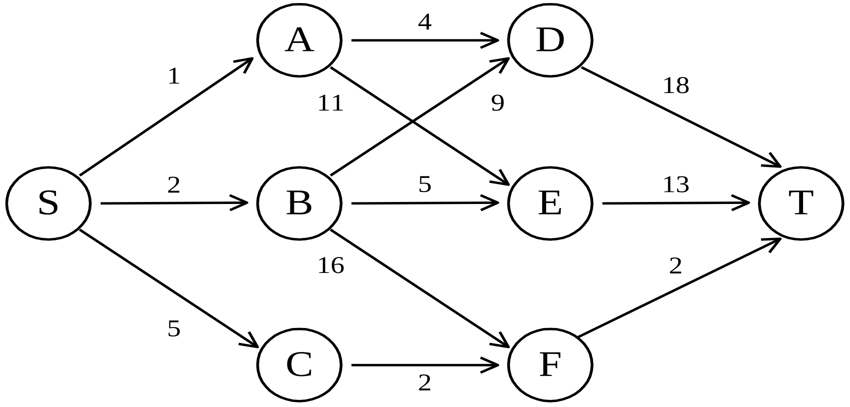
- The real shortest path is:

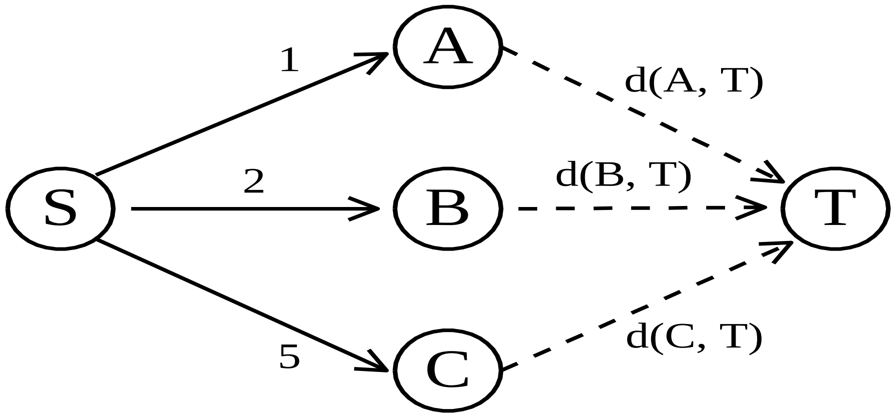
$$(S, C, F, T) = 5 + 2 + 2 = 9$$



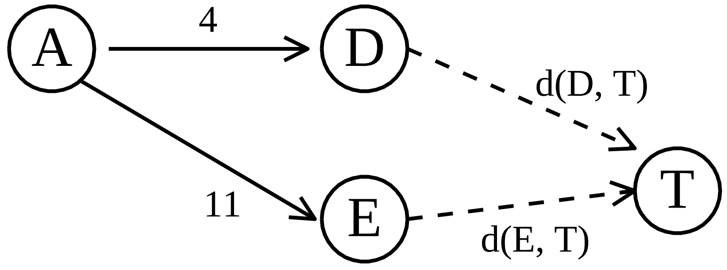
SHORTEST PATH PROBLEM

- Dynamic programming approach: Forward approach (backward reasoning):
- Recursive calls plus memoization
- $d(S,T) \; = \; \min\{1 + d(A,T), 2 + d(B,T), 5 + d(C,T)\}$



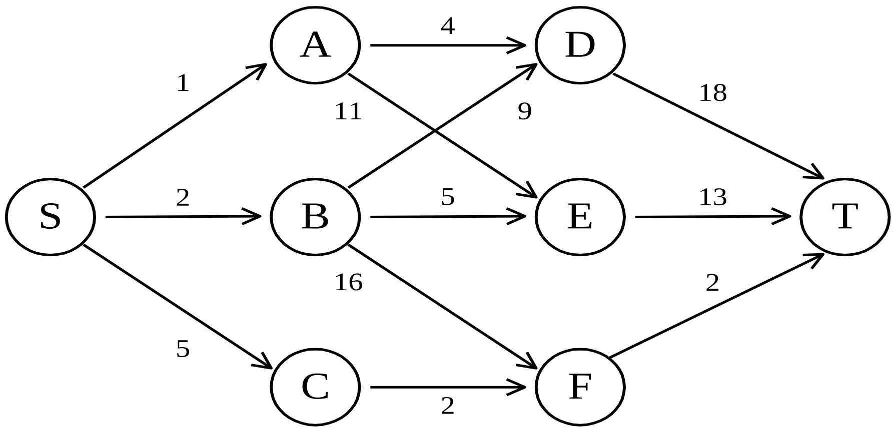


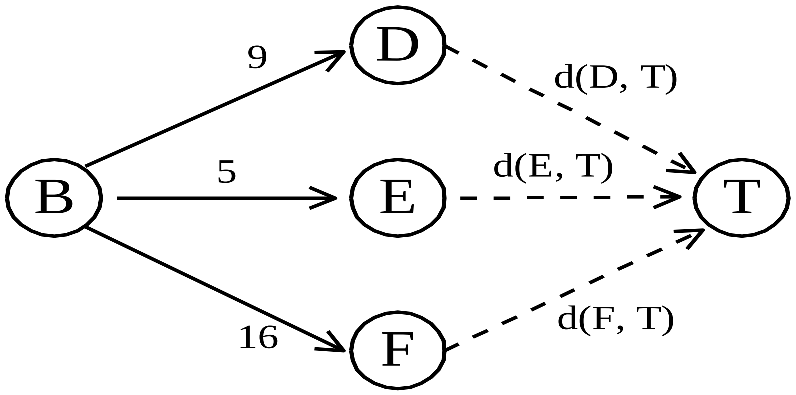
- $d(A,T) \; = \min\{4 + d(D,T), 11 + d(E,T)\}$
  $= \; \min\{4 + 18, 11 + 13\} \; = \; 22.$



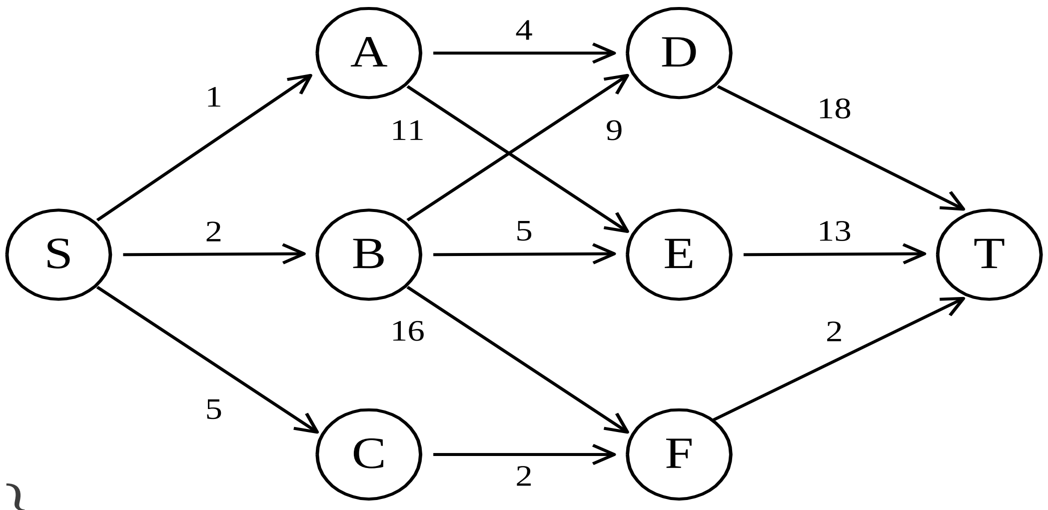
SHORTEST PATH PROBLEM

- $d(B,T) = \min\{9 + d(D,T), 5 + d(E,T), 16 + d(F,T)\}$
  $= \min\{9 + 18, 5 + 13, 16 + 2\} = 18.$
- $d(C,T) = \min\{\, 2 + d(F,T) \,\} = 2 + 2 = 4$
- $d(S,T) = \min\{1 + d(A,T), 2 + d(B,T), 5 + d(C,T)\}$
  $= \min\{1 + 22, 2 + 18, 5 + 4\} = 9.$





- Dynamic programming approach: Backward approach (forward reasoning):
- Recursive calls plus memoization



  - $d(S,A) = 1$
    $d(S,B) = 2$
    $d(S,C) = 5$

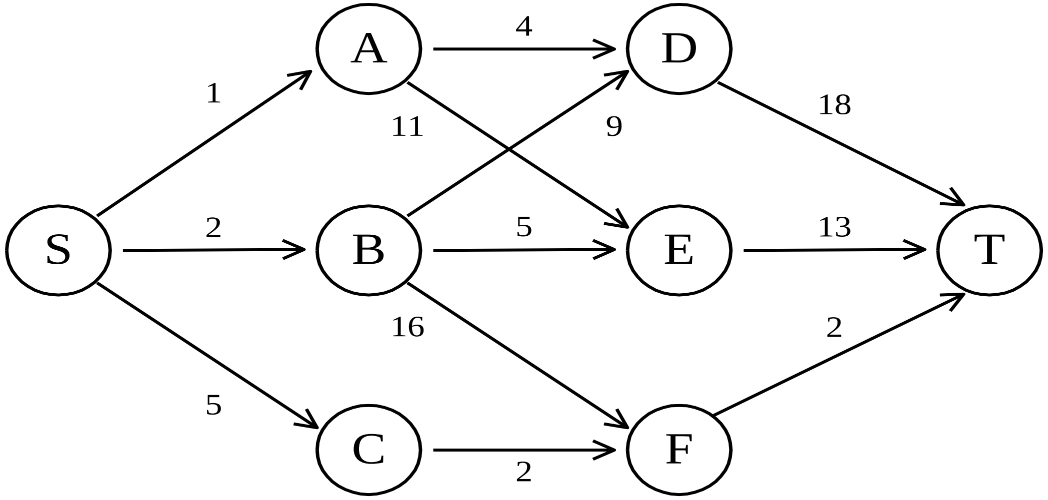
  - $d(S,D) = \min\{d(S,A) + d(A,D), d(S,B) + d(B,D)\}$
    $= \min\{ 1 + 4, 2 + 9 \} = 5$
    $d(S,E) = \min\{d(S,A) + d(A,E), d(S,B) + d(B,E)\}$
    $= \min\{ 1 + 11, 2 + 5 \} = 7$
    $d(S,F) = \min\{d(S,B) + d(B,F), d(S,C) + d(C,F)\}$
    $= \min\{ 2 + 16, 5 + 2 \} = 7$

SHORTEST PATH PROBLEM

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

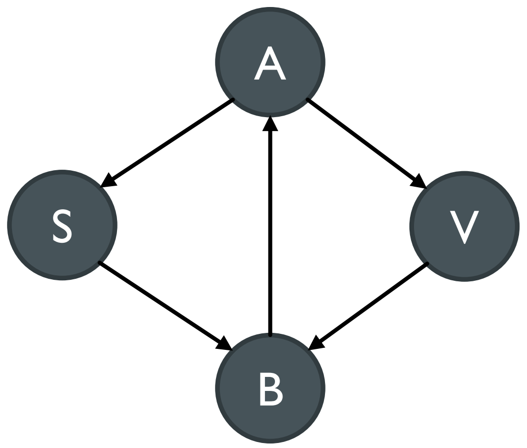
  $= \ \min\{\,5 + 18, 7 + 13, 7 + 2\,\}$

  $= \ 9$



SHORTEST PATH PROBLEM

- DP Shortest Path: Backward approach (forward reasoning):
- Number of sub-problems = $\Theta(n)$
- Time per sub-problem depends on the incoming edges ($e.g.$, degree of the vertex)
- Total Time = $O(V + E)$ (Number of sub-problems × Amount of time per sub-problem)
- Problem in the previous approach:



- $d(S,V) = d(S,A) + (A,V)$
- $d(S,A) = d(S,B) + (B,A)$
- $d(S,B) = \min\{d(S,S) + (S,B), d(S,V) + (V,B)\}$

The previous DP approach will not stop executing when the graph contains cycles as the sub-problems dependency is not acyclic.

COMMENTS

- Dynamic programming relies on saving the results of solving simpler problems
  - These solutions to simpler problems are then used to compute the solution to more complex problems
- Dynamic programming solutions can often be quite complex and tricky
- Dynamic programming is used for optimization problems, especially ones that would otherwise take exponential time
  - Only problems that satisfy the principle of optimality are suitable for dynamic programming solutions
- Since exponential time is unacceptable for all but the smallest problems, dynamic programming is sometimes essential

QUESTIONS/ANSWERS

REFERENCES

- MIT Dynamic Programming I: Fibonacci, Shortest Paths
- https://zsalloum.medium.com/how-to-think-in-dynamic-programming-3f6804a79429