Unit 5.3 The Greedy Method, III

Algorithms

EE3980

May 7, 2020

Optimal Storage on Tapes

- Given a sequentially accessed magnetic tape and n programs
  - These n programs, $1, 2, \cdots, n$, are to be stored on the tape
  - Each program has the length ℓ_i, $1 \le i \le n$.
  - The tape is always accessed from the beginning.
  - Thus, if the kth program is accessed it needs $t_k = \sum_{j=1}^{k} \ell_j$ amount of time.
  - The objective is to determined the order of the n program such that the mean retrieval time (MRT), defined as $\frac{1}{n}\sum_{k=1}^{n} t_k$, is minimum.
  - Since n is given, the minimizing MRT is equivalent to minimizing $\sum_{k=1}^{n}\sum_{j=1}^{k} \ell_{i_j}$, where i_j, $1 \le j \le n$ is a permutation of $\{1, 2, \cdots, n\}$.

Optimal Storage on Tapes — Example

- Example
  - $n = 3$ and $\{\ell_1, \ell_2, \ell_3\} = \{5, 10, 3\}$.
  - There are 6 permutations all of which are feasible solutions.

Ordering	$\sum_{k=1}^{n}\sum_{j=1}^{k} \ell_{i_j}$	
1,2,3	5+(5+10)+(5+10+3)	= 38
1,3,2	5+(5+3)+(5+3+10)	= 31
2,1,3	10+(10+5)+(10+5+3)	= 43
2,3,1	10+(10+3)+(10+3+5)	= 41
3,1,2	3+(3+5)+(3+5+10)	= 29
3,2,1	3+(3+10)+(3+10+5)	= 34

- The optimal ordering is $\{3, 1, 2\}$.

Optimal Storage on Tapes — Optimality and Complexity

- Note that the objective function can be written as

$$\sum_{k=1}^{n}\sum_{j=1}^{k} \ell_{i_j} = (\ell_{i_1}) + (\ell_{i_1} + \ell_{i_2}) + (\ell_{i_1} + \ell_{i_2} + \ell_{i_3}) + \cdots$$
$$= n\ell_{i_1} + (n-1)\ell_{i_2} + (n-2)\ell_{i_3} + \cdots$$

- Thus ℓ_{i_1} should be the smallest possible to reduce MRT
- Once i_1 is determined, ℓ_{i_2} should be the smallest among the remaining programs.

Theorem 5.3.1.

If $\ell_1 \le \ell_2 \le \cdots \le \ell_n$, then the ordering i_j, $1 \le j \le n$, minimizes

$$\sum_{k=1}^{n}\sum_{j=1}^{k} \ell_{i_j} \tag{5.1}$$

over all possible permutation of i_j.

- Thus, the optimal storage on tape problem reduces to the ordering of the n programs by their lengths — $\mathcal{O}(n \lg n)$.

Optimal Storage on Tapes — Multi-tape Case

- The number of tapes can be m, $m \geq 1$
- The program should be distributed over the m tapes
- The following algorithm assigns the n programs to m tapes that achieves minimum MRT.

Algorithm 5.3.2. Multi-tape Storage

```
  // Store n programs, each has length ℓ[1 : n], onto m tapes.
  // Input: int n, m, ℓ[1 : n]
  // Output: Program storage assignments.
1 Algorithm store(n, l, m)
2 {
3     Sort(l) in non-decreasing order ;
4     j := 1; // Next tape to store on
5     for i := 1 to n do {
6         Append program i to tape j;
7         j := (j + 1) mod m;
8     }
9 }
```


Multi-tape Storage — Complexity and Optimality

- Note that the time complexity of Algorithm (5.3.2) is dominated by line 3 Sort function, which has time complexity of $\mathcal{O}(n \lg n)$.

Theorem 5.3.3.

If $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_n$, then Algorithm (5.3.2) generates an optimal storage pattern for m tapes.

- Proof see textbook [Horowitz], pp. 251 – 252.
- Note that there can be more than one optimal assignment if some program lengths are equal.

Merging Multi-Files

- Merging two files containing n and m records need to move $n + m$ data.
- Let's consider two-way merge pattern only, i.e., merge two files each time.
- Given multiple files with different number of records, what is the order of binary merge to achieve minimum number of moves.

- Example
- 3 sorted files x_1, x_2 and x_3 with 30, 20 and 10 data each.
  1. Merge x_1 and x_2 first requires 50 moves;
     Then merge with x_3 requires another 60 moves;
     Total number of moves is 110.
  2. Merge x_2 and x_3 first in 30 moves;
     Then merge with x_1 in 60 moves;
     Total number of moves is 90.
- Observation: to merge smaller files first.

Merging Multi-Files — Algorithm

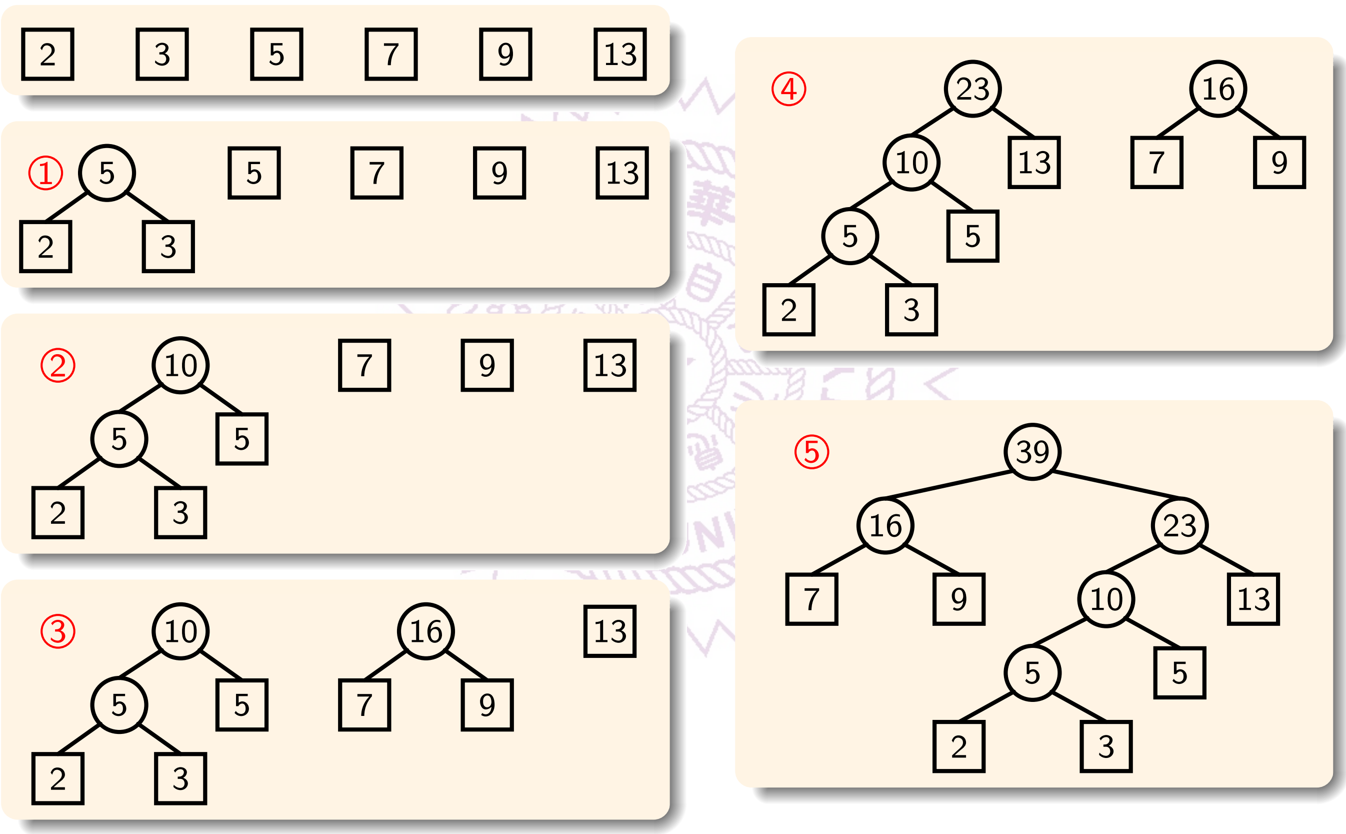
- Using the node structure as

```
struct node {
      struct node *lchild, *rchild;
      integer w;
}
```

Algorithm 5.3.4. Binary Merge Tree

```
// Generate binary merge tree from list of n files.
// Input: int n, list of files
// Output: optimal merge order.
Algorithm Tree(n, list)
{
      for i := 1 to (n − 1) do {
            pt := new node;
            pt → lchild := Least(list); // Find and remove min from list.
            pt → rchild := Least(list);
            pt → w := (pt → lchild) → w + (pt → rchild) → w;
            Insert(list, pt);
      }
      return Least(list);
}
```

Merging Multi-Files — Example



Merging Multi-Files — Properties

- Two functions are used the `Tree` algorithm
  - `Least` finds and removes the smallest data item from *list*,
  - `Insert` inserts the tree *pt* to the *list*.
- In the preceding example
  - Data files are sorted by their sizes and arranged in a simple list initially.
  - A two-way merge is then applied for the first two data files.
    - A tree is created with the data files as leaves – also called external nodes, shown in squares.
    - A new node, an internal node, is created with sum of its children as its weight, shown in a circle.
  - At the end, a binary tree is obtained.
  - For an external node with size q_i at level i of the binary tree
    - Its distance to the root is $d_i = i - 1$.
    - And it contributes $d_i q_i$ moves to the total number of moves.
    - And the total number of moves of the merge operations is

$$\sum_{i=1}^{n} d_i q_i \tag{5.2}$$

This sum is called the weighted external path length of the tree.

Merging Multi-Files — Complexity and Optimality

- In Algorithm (5.3.4), the `while` loop is executed $n-1$ times.
- If the *list* is kept in non-decreasing order, then
  - `Least` takes $\mathcal{O}(1)$ time,
  - And `Insert` takes $\mathcal{O}(n)$ time,
  - Thus, the overall time complexity is $\mathcal{O}(n^2)$.
- If the *list* is represented by a `minheap` then
  - Both `Least` and `Insert` can be done in $\mathcal{O}(\lg n)$ time,
  - The overall time complexity is $\mathcal{O}(n \lg n)$.

Theorem 5.3.5.

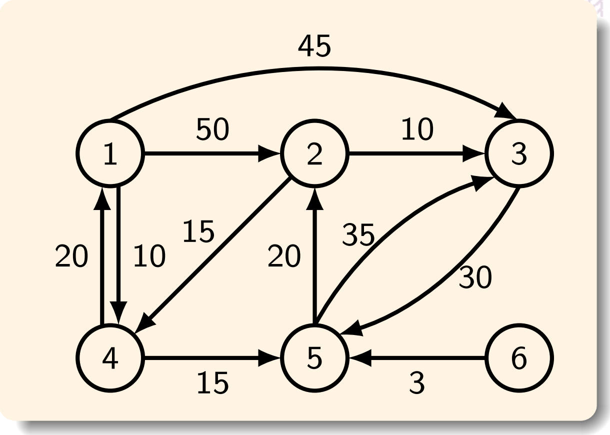
If the *list* initially contains $n \geq 1$ single node trees with weight values $\{q_1, q_2, \cdots, q_n\}$, then the `Tree` algorithm (5.3.4) generates an optimal two-way merge tree for n files with these lengths.

- Proof see textbook [Horowitz], p. 257.

- The two-way merge can be generalized to k-way merge problems.
- Huffman code is an application of two-way merge method.

Single-Source Shortest Paths

- Given a directed graph $G = (V, E)$, a weight function on the edges in E, $w : E \to \mathbb{R}$, and source vertex v_0, the single-source shortest path problem is to determine the shortest paths from v_0 to all remaining vertices.
- The weight of a path $P = \langle v_1, v_2, \ldots, v_k \rangle$ is the sum of the weights of the edges, $w(P) = \sum_{k=1}^{k-1} w(v_k, v_{k+1})$.
- Define $\delta(s, v) = \min\{w(P) | P \text{ is a path from } s \text{ to } v\}, s, v \in V$.
- The problem is to find $\delta(s, v)$ for all $v \in V$.
- Example



$v_0 = 1$

	Path	Length
1	1,4	10
2	1,4,5	25
3	1,4,5,2	45
4	1,3	45

Single-Source Shortest Paths – Properties

Lemma 5.3.6. Subpaths of shortest paths are shortest paths

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, if $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i < j \le k$, $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a subpath from vertex i to vertex j, then p_{ij} is a shortest path from v_i to v_j.

- Proof please see textbook [Cormen], p. 645.

- In this section, the weight of an edge is assumed to be non-negative.
- Thus, the weight of any cycle is also non-negative.
- A shortest path should not include any cycle, since the cycle can be removed to obtain a shorter path.
- Therefore, any shortest paths has at most $n - 1$ edges, $n = |V|$.

Single-Source Shortest Paths – Algorithm

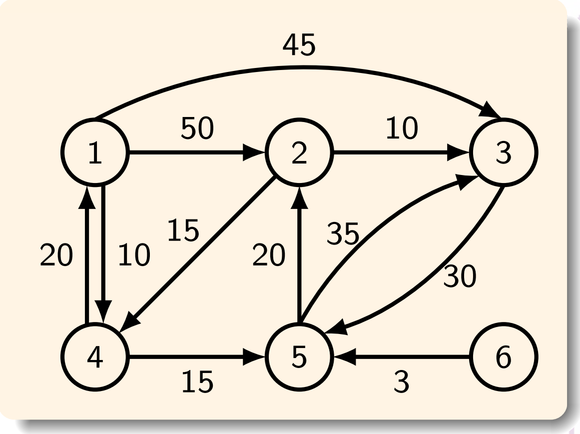
Algorithm 5.3.7. Dijkstra's Algorithm

```
   // Find the shortest paths from v and fill the path lengths to d[1 : n] array.
   // Input: int n, node v, weight w[1 : n]
   // Output: distance array d[1 : n].
Algorithm ShortestPaths(n, v, w, d)
{
      for i := 1 to n do {
            S[i] := false ;
            d[i] := w[v, i] ;
      }
      S[v] := true ;
      d[v] := 0 ;
      for k := 2 to n do {
            Find i such that S[i] = false and d[i] is minimum ;
            S[i] := true ;
            for ( each j adjacent to i and S[j] = false ) do {
                  if (d[j] > d[i] + w[i, j]) then
                        d[j] := d[i] + w[i, j] ;
            }
      }
}
```

- $S[1 : n]$ is an array to indicate if the shortest path for a vertex has been found or not.

Single-Source Shortest Paths – Example

- Given the graph on the left, the shortest paths to all other vertices are found.



Vertex		1	2	3	4	5	6
k=1	S	1	0	0	0	0	0
	d	0	50	45	10	∞	∞
k=2	S	1	0	0	1	0	0
	d	0	50	45	10	25	∞
k=3	S	1	0	0	1	1	0
	d	0	45	45	10	25	∞
k=4	S	1	1	0	1	1	0
	d	0	45	45	10	25	∞
k=5	S	1	1	1	1	1	0
	d	0	45	45	10	25	∞
k=6	S	1	1	1	1	1	0
	d	0	45	45	10	25	∞

- Note that to print out the shortest path for each vertex, an additional array, $p[1:n]$, to record the predecessor of the path is needed and line 12 should be modified to add $p[j] := i$.

Single-Source Shortest Paths – Complexity

- Algorithm (5.3.7) is dominated by the for loop in lines 7-14.
  - This loop executes $(n-1)$ times.
  - Line 8 takes $\mathcal{O}(n)$ time,
  - The for loop on Lines 10-13 takes $\mathcal{O}(n)$ time,
  - The overall complexity is $\mathcal{O}(n^2)$.
- The time complexity of the algorithm can be improved to $\mathcal{O}((n+|E|)\lg n)$ with proper data structures.
- Algorithm (5.3.7) generates the shortest paths from vertex v to all other vertices in G.
- The edges of the shortest paths from a vertex v to all other vertices in a connected undirected graph G form a spanning tree – shortest-path spanning tree.
  - Different source vertex can have different spanning tree.
  - This tree can also be different from the minimum-cost spanning tree.

Single-Source Shortest Paths – Correctness

Theorem 5.3.8.

Given a weighted, directed graph $G = (V, E)$ with non-negative weight function w and a source vertex v, Algorithm (5.3.7) produces $d[u] = \delta(s, u)$ for all vertices $u \in V$.

- Proof please see textbook [Cormen], p. 660-661.
- As a corollary of the above theorem, if the predecessor array $p[1:n]$ is also implemented in Algorithm (5.3.7) then the solutions printed using array p are the shortest paths from vertex v.

Single-Source Shortest Paths – Directed Acyclic Graphs

- A directed acyclic graph (DAG) $G = (V, E)$ is a directed graph without any cycles.
- Since no cycle exists, the non-negative weight function constraint can be relaxed – no negative cycle possible.
- In this case, the following algorithm is effective in finding the shortest path

Algorithm 5.3.9. Shortest path for DAG

```
// Find the shortest paths from v and fill the path lengths to d[1 : n] array.
// Input: int n, node v, weight w[1 : n]
// Output: distance array d[1 : n].
Algorithm ShortestPaths_DAG(n, v, w, d)
{
    Let slist[1 : n] be the topological sort of the directed acyclic graph ;
    d[v] := 0 ;
    for i := 1 to n do {
        for ( each j adjacent to slist[i]) do {
            if (d[j] > d[i] + w[i, j]) then
                d[j] := d[i] + w[i, j] ;
        }
    }
}
```

DAG Single-Source Shortest Paths

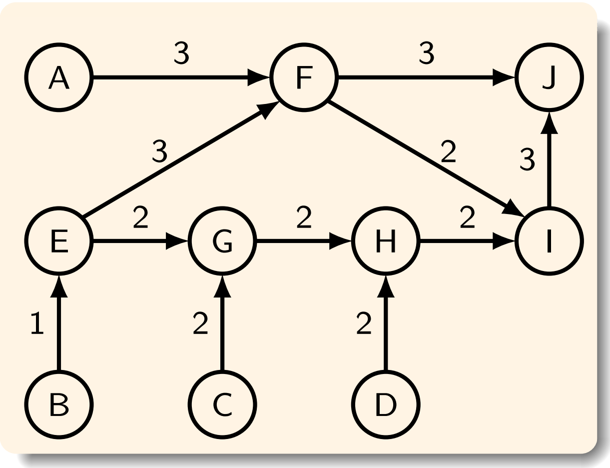
- The complexity of the algorithm
  - The topological sort, line 3, has the complexity $\mathcal{O}(n+e)$
    - $n = |V|$, $e = |E|$
  - The `if` statement, lines 7-8, executes e times
  - The overall complexity is $\mathcal{O}(n+e)$.

Theorem 5.3.10.

Given a directed acyclic graph $G = (V, E)$, algorithm (5.3.9) produces $d[v] = \delta(s, v)$, $v \in V$.

- Proof please see textbook [Cormen], pp. 656-657.
- The shortest path can be printed if the predecessor array is also kept.

DAG Single-Source Shortest Paths – Example

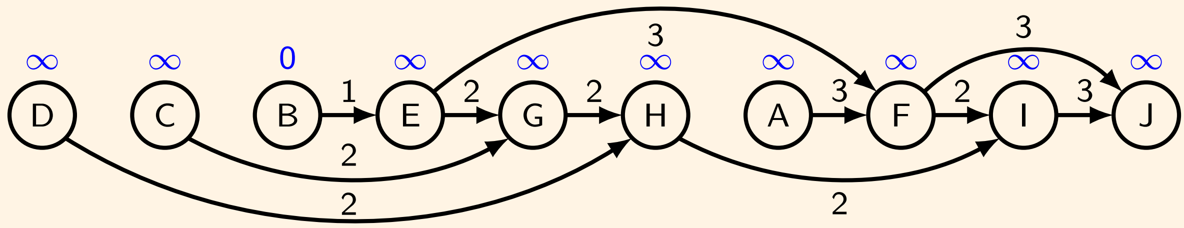


- Given a weighted DAG above, if vertex B is the source we have the shortest path length for each vertex below.

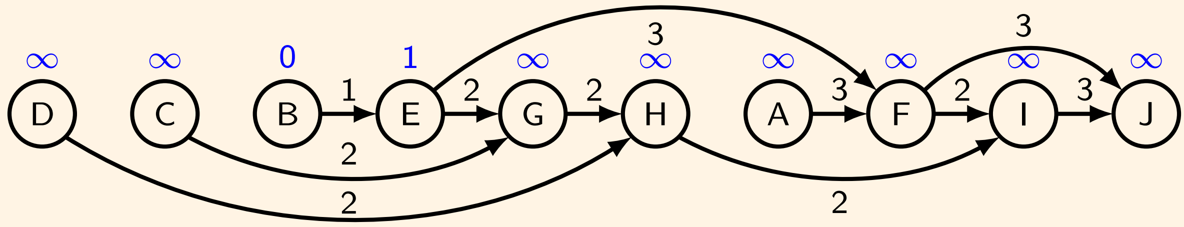
Vertex	A	B	C	D	E	F	G	H	I	J
δ	∞	0	∞	∞	1	4	3	5	6	7

DAG Single-Source Shortest Paths – Example, II

- Execution sequences of Algorithm (5.3.9) is shown below
- After line 4:

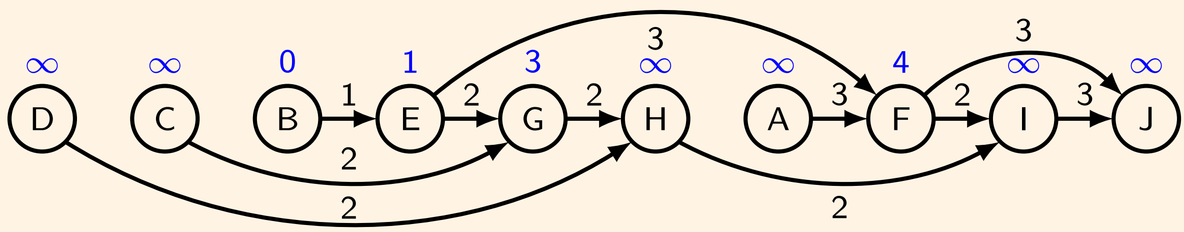


- In the `for` loop, $i = 3$

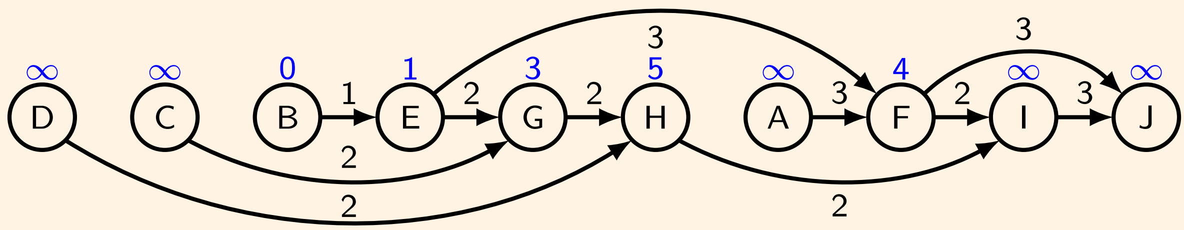


DAG Single-Source Shortest Paths – Example, III

- In the `for` loop, $i = 4$

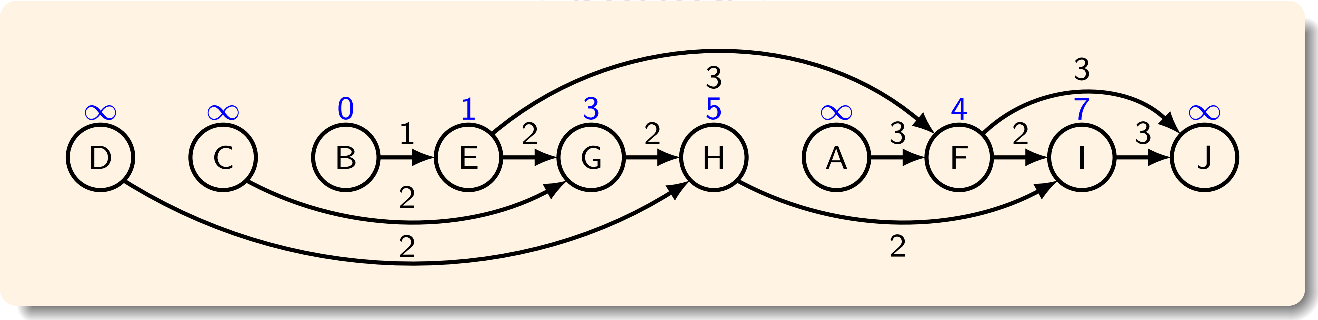


- In the `for` loop, $i = 5$

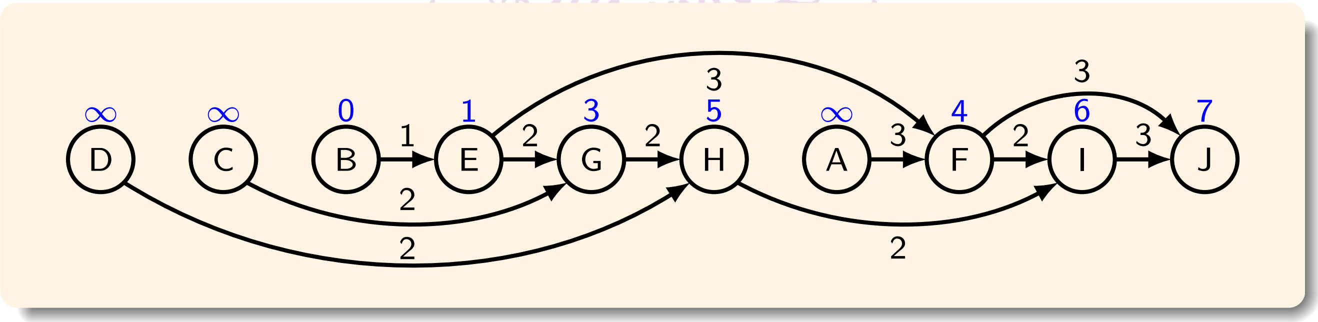


DAG Single-Source Shortest Paths – Example, IV

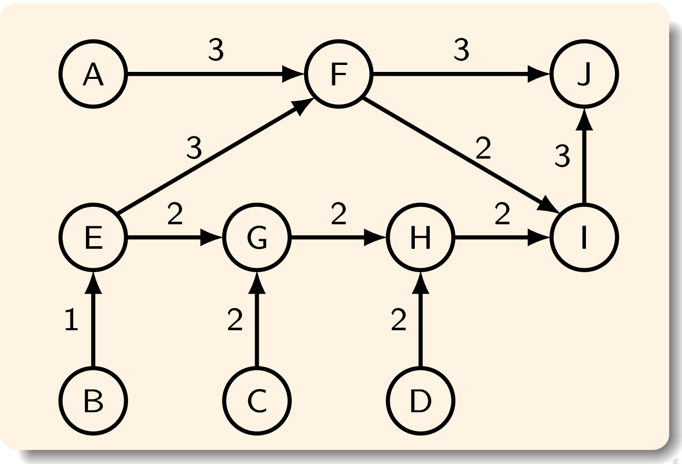
- In the `for` loop, $i = 6$



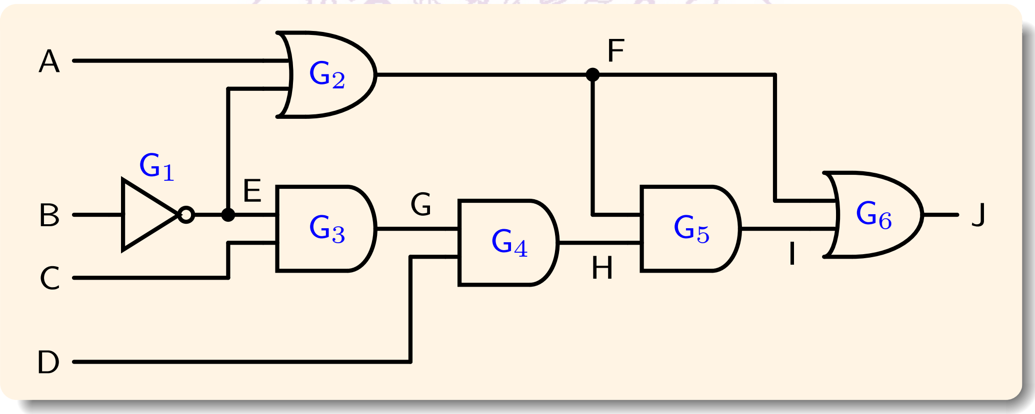
- In the `for` loop, $i = 8$



DAG Single-Source Shortest Paths – Application



- The weighted direct graph is actually the digital circuit delay path, and the shortest path represent the delay from input B to various nodes.
- INV delay = 1, ND2 delay = 2, NR2 delay = 3.



Summary

- Optimal storage on tapes.
- Optimal merge patterns.
- Single-source shortest path.