CS/ECE 374 Spring 2025 Heng An Cheng (hacheng2@illinois.edu)
Homework 8 Problem 1

Solution:

(a)

Denote $MWVC(v, j) =$ minimum weight of a vertex cover in the subtree rooted at v using exactly j vertices,

with the convention that if no such cover exists then $MWVC(v, j) = +\infty$.

Base Case: For a leaf v,

$$MWVC(v, 0) = 0, \quad MWVC(v, 1) = w(v), \quad MWVC(v, j) = +\infty \quad \text{for } j > 1.$$

Recursive Step: For an internal vertex v with children $u_1, u_2, \ldots, u_m$,

$$MWVC(v, j) = \min \left\{ \begin{array}{l} \min\limits_{j_1+\cdots+j_m=j-1} \Big[w(v) + \sum\limits_{i=1}^{m} MWVC(u_i, j_i) \Big], \\ \min\limits_{\substack{j_1+\cdots+j_m=j \\ j_i \geq 1 \, \forall i}} \Big[\sum\limits_{i=1}^{m} MWVC(u_i, j_i) \Big] \end{array} \right\}.$$

The first term corresponds to the case where v is included in the cover, and the second term corresponds to the case where v is not included (forcing every child u_i to be included).

Final Answer: Since we want a vertex cover of size *at most* k, the answer for the tree (with root r) is

$$\min_{0 \leq j \leq k} MWVC(r, j).$$

Time Analysis

For each vertex v, we compute $MWVC(v, j)$ for $j = 0, \ldots, k$. By merging the DP tables of the children in $O(k^2)$ time per merge, if v has at most d children, the processing time per vertex is $O(d \cdot k^2)$. Thus, over all n vertices, the total time is

$$O(n \cdot d \cdot k^2).$$

If d is a constant, this simplifies to $O(n \cdot k^2)$.

(b) (i) Let $T = (V, E)$ be a tree with an arbitrary root r. We define two functions for every vertex v:

$f(v) =$ the number of vertex covers of the subtree rooted at v that include v,
$g(v) =$ the number of vertex covers of the subtree rooted at v that exclude v.

Thus, the total number of vertex covers for the subtree rooted at v is:

$$C(v) = f(v) + g(v).$$

Base Case: If v is a leaf (i.e. v has no children), then the vertex cover on v can either include v or not, giving:

$$f(v) = 1, \quad g(v) = 1, \quad \text{so } C(v) = 2.$$

Recursive Step: For an internal vertex v with children $u_1, u_2, \ldots, u_m$, we have two cases:

- **Case 1: v is included in the cover.** Since v covers all edges (v, u_i), each child u_i can be either in or out of the cover. Hence, the number of vertex covers in this case is:
$$f(v) = \prod_{i=1}^{m} \Big(f(u_i) + g(u_i) \Big).$$
- **Case 2: v is not included in the cover.** If v is excluded, then to cover the edge (v, u_i) each child u_i *must* be included in the cover. Thus:
$$g(v) = \prod_{i=1}^{m} f(u_i).$$

Final Answer: The total number of distinct vertex covers for the entire tree T is given by:

$$C(r) = f(r) + g(r).$$

Time Analysis

Each vertex $v \in v$ is processed exactly once in a postorder traversal of the tree. For each vertex, we perform a constant amount of work (multiplying the results from its children). Hence, the overall time complexity of the algorithm is $O(|v|)$.

(ii) We consider a path P_n on n nodes. For a single node (i.e. $n = 1$) the vertex cover can be either the empty set (which is valid since there are no edges) or the singleton set containing the node. Thus, the base case is:

$$DP[1] = 2.$$

For convenience, we also define a base case for the empty path (no nodes) as:

$$DP[0] = 1.$$

Now, for $n \geq 2$, consider the rightmost node (say node n):

- **Case 1:** Node n is *included* in the vertex cover. In this case, the remaining $n - 1$ nodes can be covered in any valid way. This contributes $DP[n-1]$ covers.
- **Case 2:** Node n is *not included*. Then to cover the edge $(n-1, n)$, node $n-1$ *must* be included. In this case, the first $n-2$ nodes can be covered in any valid way. This contributes $DP[n-2]$ covers.

Thus, the recurrence is:

$$DP[n] = DP[n-1] + DP[n-2], \quad \text{for } n \geq 2.$$

This recurrence is exactly that of the Fibonacci sequence. To see the correspondence, define a Fibonacci sequence with the following initial conditions:

$$F[0] = 1, \quad F[1] = 1, \quad \text{and } F[n] = F[n-1] + F[n-2] \text{ for } n \geq 2.$$

Then, our recurrence satisfies:

$$DP[0] = F[1] = 1, \quad DP[1] = 2 = F[2],$$

and by induction one can show that

$$DP[n] = F[n+1].$$

Fitting in a 64-bit Word: The Fibonacci numbers grow exponentially; indeed, using Binet's formula we have

$$F[n] \approx \frac{\varphi^n}{\sqrt{5}},$$

where $\varphi \approx 1.618$. For $n = 501$, we have

$$DP[500] = F[501] \approx \frac{\varphi^{501}}{\sqrt{5}}.$$

Taking logarithms (base 10):

$$\log_{10} F[501] \approx 501 \cdot \log_{10} \varphi - \tfrac{1}{2} \log_{10} 5 \approx 501 \cdot 0.208987 - 0.349 \approx 104.44.$$

Thus, $F[501]$ has roughly 105 decimal digits. A 64-bit integer can hold values up to roughly $2^{63} \approx 9.22 \times 10^{18}$, which is only about 19 decimal digits. Therefore, the answer for a path with $n = 500$ does *not* fit in a 64-bit integer.

(iii) **Implementation:** Use multi-precision arithmetic by representing large integers as arrays of 64-bit words. Each addition is performed word-by-word with carry propagation. For example:

- Use an array (or vector) where each entry stores a 64-bit word representing a "digit" in a fixed base (2^{64}).
- Implement addition on these arrays by performing word-level additions with carry propagation.

Running Time: There are n steps, and each addition takes $O(n)$ machine-word operations (since numbers have $O(n)$ bits), yielding an overall running time of $O(n^2)$.

■

CS/ECE 374 Spring 2025 Heng An Cheng (hacheng2@illinois.edu)
Homework 8 Problem 2

Solution:

(a) The algorithm consists of two main steps:

- **DFA Construction:** We construct a deterministic finite automaton (DFA) $D = (Q, \Sigma, \delta, q_0, F)$ that accepts exactly those strings which contain the substring "374". The alphabet Σ is $\{0, 1, \ldots, 9\}$. The states of D represent the progress in matching the pattern "374". For example, let

$$Q = \{q_0, q_1, q_2, q_3\},$$

where:

- q_0 is the start state (no characters matched),
- q_1 indicates that 3 has been matched,
- q_2 indicates that 37 has been matched,
- q_3 is the accepting state meaning that 374 has been matched.

The transition function δ is defined appropriately to update the match progress based on the next digit.

- **Product Graph and DFS:** We now construct the product graph $G' = (V', E')$ where:

$$V' = V \times Q.$$

There is an edge from (u, q) to (v, q') in E' if and only if $(u, v) \in E$ in G and $q' = \delta(q, \ell(v))$. We then perform a DFS (or BFS) starting from the initial state (s, q_0) in the product graph and check if we can reach a vertex of the form (t, q_3) (i.e., t paired with the accepting state q_3).

Algorithm 1 Find an s-t walk with substring `374`

Input: Graph $G = (V, E)$, labeling function $\ell : V \to \{0, 1, \ldots, 9\}$, vertices s, t
Output: `true` if there exists an s-t walk W in G with $\sigma(W)$ containing `374`, else `false`
Construct DFA $D = (Q, \Sigma, \delta, q_0, F)$ where $Q = \{q_0, q_1, q_2, q_3\}$ and $F = \{q_3\}$.
Initialize stack $S \leftarrow \{(s, q_0)\}$
Initialize visited set $Visited \leftarrow \{(s, q_0)\}$
while S is not empty **do**
    $(u, q) \leftarrow$ pop from S
    if $u = t$ and $q = q_3$ **then**
        return `true`
    end if
    for each edge $(u, v) \in E$ **do**
        $q' \leftarrow \delta(q, \ell(v))$
        if $(v, q') \notin Visited$ **then**
            Add (v, q') to $Visited$
            Push (v, q') onto S
        end if
    end for
end while
return `false`

Time Analysis

- The DFA D has a constant number of states (4 in this case) and can be constructed in constant time.
- The product graph G' has $O(|V|)$ vertices (specifically, $|V| \times 4$) and each edge $(u, v) \in E$ creates at most 4 transitions in G'. Thus, the total number of transitions is $O(|E|)$.
- The DFS on the product graph runs in $O(|V'| + |E'|)$ time, which is $O(|V| + |E|)$.

Thus, the overall algorithm runs in $O(|V| + |E|)$ time.

(b) We utilize the same DFA and product graph G' from (a) if there exists an s-t walk W in G with $\sigma(W)$ containing 374.

The algorithm proceeds in the following steps:

- **Reverse Reachability Search:** We wish to know for which vertices v there exists a walk from (v, q_0) to (t, q_3) in G'. Instead of searching from every possible v, we reverse the product graph and perform a DFS (or BFS) starting from the target state (t, q_3). Let R be the set of all product states that can reach (t, q_3) in the reverse graph.
- **Output the Answer:** The final answer is the set of all vertices $v \in V$ for which (v, q_0) is in R.

Algorithm 2 Output set of vertices with a v-t walk containing substring 374

Input: Graph $G = (V, E)$, labeling function $\ell : V \to \{0, 1, \ldots, 9\}$, vertex t
Output: Set $S \subseteq V$ of all vertices v such that there exists a v-t walk W_v with $\sigma(W_v)$ containing 374

/* Step 1: Construct DFA for substring 374 */
Define $Q = \{q_0, q_1, q_2, q_3\}$ with start state q_0 and accepting state $F = \{q_3\}$
Define transition function $\delta : Q \times \{0, 1, \ldots, 9\} \to Q$ (details omitted for brevity)

/* Step 2: Build the product graph implicitly */
Let $V' = \{(v, q) : v \in V, q \in Q\}$
/* We do not explicitly construct E'; instead we use δ on the fly */

/* Step 3: Reverse reachability from (t, q_3) */
Initialize stack $S \leftarrow \{(t, q_3)\}$
Initialize visited set $Visited \leftarrow \{(t, q_3)\}$

while S is not empty **do**
    Pop (u, q) from S
    for each edge $(v, u) \in E$ **do** ▷ Reverse edge: u is reached from v
        Compute q' such that $\delta(q', \ell(u)) = q$ ▷ (Determine predecessor DFA state q' given that reading $\ell(u)$ leads to q.)
        if $(v, q') \notin Visited$ **then**
            Add (v, q') to $Visited$
            Push (v, q') onto S
        end if
    end for
end while

/* Step 4: Collect all vertices v with (v, q_0) in $Visited$ */
$Result \leftarrow \{v \in V : (v, q_0) \in Visited\}$
return $Result$

Time Analysis

- **DFA Construction:** The DFA has a constant number of states (4) and is constructed in $O(1)$ time.
- **Product Graph:** The product graph G' has $|V| \times 4 = O(|V|)$ vertices. Each edge $(u, v) \in E$ yields a transition in the product graph. Thus, there are $O(|E|)$ transitions.
- **Reverse DFS:** The reverse reachability search traverses at most all product vertices and edges, in time $O(|V'| + |E'|) = O(|V| + |E|)$.
- **Overall Complexity:** The algorithm runs in $O(|V| + |E|)$ time.

(c) We construct a new (configuration) graph whose vertices are triples (u, v, w) with u, v, w being distinct vertices of G. There is an edge from (a, b, c) to (a', b', c') if and only if

$$a' \in N(a), \quad b' \in N(b), \quad c' \in N(c),$$

and a', b', c' are all distinct. Here, $N(a)$ denotes the *neighborhood* of the vertex a in G, i.e., the set of all vertices that are adjacent to a in G.

We then require that the tokens only trigger the printouts at the correct times by dividing the motion into three phases:

- **Phase 1:** From (s_1, s_2, s_3) to $(1, 7, 3)$ while *avoiding* $(3, 7, 4)$.
- **Phase 2:** From $(1, 7, 3)$ to $(3, 7, 4)$ while *avoiding* any visit to $(1, 7, 3)$ or $(3, 7, 4)$ except at the endpoints.
- **Phase 3:** From $(3, 7, 4)$ to (t_1, t_2, t_3) while avoiding any visit to $(1, 7, 3)$ or $(3, 7, 4)$.

A path with these properties will produce the printed string `173` upon entering phase 1's target state, then `374` upon finishing phase 2, and no further printed substrings during phase 3.

Algorithm 3 Check for a Valid Sequence Producing `173374`

procedure ValidPath(G, $s = (s_1, s_2, s_3)$, $t = (t_1, t_2, t_3)$)
  Let $S = \{(u, v, w) \mid u, v, w \text{ distinct vertices of } G\}$
  Construct configuration graph $H = (S, E)$ where

Edge $((a, b, c), (a', b', c')) \in E \iff a' \in N(a)$, $b' \in N(b)$, $c' \in N(c)$ and a', b', c' are distinct.

  Define special states:

$$p = (1, 7, 3) \quad \text{and} \quad q = (3, 7, 4).$$

  // **Phase 1:** Find a path in H from s to p avoiding q.
  if BFS(H, s, p, forbidden=$\{q\}$) returns `false` **then**
    return `false`
  end if
  // **Phase 2:** Find a path in H from p to q with no intermediate occurrence of p or q.
  if BFS(H, p, q, forbidden=$\{p, q\} \setminus \{p, q\}$) returns `false` **then**
    return `false`
  end if
  // **Phase 3:** Find a path in H from q to t avoiding $\{p, q\}$.
  if BFS(H, q, t, forbidden=$\{p, q\}$) returns `false` **then**
    return `false`
  end if
  return `true`
end procedure

Here, BFS is a breadth-first search procedure modified so that it does not traverse any vertex in the given `forbidden` set (except when the start or target is itself in that set, as allowed in the endpoints for phase 2).

Let n be the number of vertices in G. In the configuration graph H:

- The number of vertices is $V = O(n^3)$ (since each state is a triple of distinct vertices).
- Each vertex has at most $O(\Delta^3)$ neighbors, where Δ is the maximum degree of G. Hence, the number of edges is

$$E = O(n^3 \cdot \Delta^3).$$

Since a BFS runs in $O(V + E)$ time, one BFS takes

$$O(n^3 + n^3\Delta^3) = O(n^3\Delta^3).$$

As we perform a constant number (three) of BFS operations, the overall time complexity of the algorithm is

$$O(n^3\Delta^3).$$

If Δ is bounded by a constant, this further simplifies to $O(n^3)$.

■