Graphs **May 28, 2025**

Homework 7

Spring 2025 *COS31: Algorithms*

1. **Priority Queue**
Suppose that $a[1 \dots n]$ is an array, where each $a[i]$ is a pair consisting of a key and a value. In CS 10 you studied a data structure called a binary heap, which can be used to store the (key, value) pairs in the array a. As you might recall, the heap data structure enables efficient implementation of the following operations

Answer

- $insert(a, i)$ has $O(\log n)$
- $findmin()$: has $O(1)$
- $deletemin()$ has $O(\log n)$
- $decrease - key(i, k)$ has $O(\log n)$

2. **Graph transposition**
Write pseudocode for a simple method that, given n and the adjacency list representation of a directed graph $G = (V, E)$ of n vertices, returns the adjacency list representation of GT , the transpose of G, defined by (V, E^T), where $E^T = \{(u, v)|(v, u) \in E\}$., State the time complexity. You merely have to state the algorithm and its time complexity, and there is no need to justify correctness.

Answer

Algorithm Graph transposition

Pre-condition: $n \leq 1$, a is an array containing the adjacency list of a graph $G = (V, E)$
Postcondition: Terminates, and returns an adjacency list of G' which is a transposition of G

```
Function GraphTransposition (a[1...n]):
    init A[1...n] to store transposed
    for i ← 1 to n do
        curr ← a[i]
        while curr ≠ NULL do
            A[curr.val].append(a[i])
            curr ← curr.next
    return A
```

Time complexity: $O(n + m)$ where $n = |V|$ and $m = |E|$
Space complexity: $O(n)$, $n = |V|$

3. **Cycle**
Give an algorithm that, given an undirected graph $G = (V, E)$, outputs whether or not G contains a cycle. I require that your algorithm has a time complexity of $O(n)$, independent of m. Explain why your algorithm's time complexity is $O(n)$. You don't need to prove your algorithm correct.

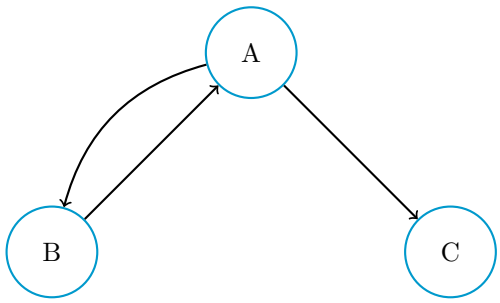
Answer

Algorithm Cycle detection (Undirected Graph with Colors)

```
Function CycleDetection(G(V, E)):
    foreach u ∈ V do
        u.c ← white
        u.π ←⊥
    foreach u ∈ V do
        if u.c = white and DFSVisit(u) = true then
            return true
    return false
Function DFSVisit(u):
    u.c ← grey
    foreach v ∈ Adj[u] do
        if v.c = white then
            v.π ← u
            if DFSVisit(v) = true then
                return true
        else if v.c = grey and v ≠ u.π then
            return true
    u.c ← black
    return false
```

Time complexity: The DFS explores each vertex once and each edge at most twice. In the worst case, if the graph has no cycles, then it must be a forest (collection of trees), where $|E| \leq |V| - 1$. Thus, the total work is $O(|V| + |E|) = O(n)$. If a cycle is found early, the DFS halts, and until that point, the visited subgraph is also a tree — meaning the number of traversed edges is still at most $|V| - 1$, giving $O(n)$ time.

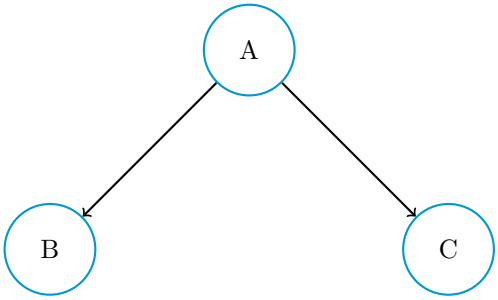
4. **DFS Counterexamples**
Give a counterexample to each of the following statements. As you know, whenever you are asked to give a counterexample, your answer should meet three criteria: (i) you should give a counterexample, (ii) you should explain why it is a counterexample, and (iii) your counterexample should be as simple as possible.



(a) If a directed graph G contains a path from u to v and if $u.d < v.d$ in a depth-first search of G, then v is a descendent of u in the depth-first forest produced.

Answer

Consider the directed graph G with vertices $\{A, B, C\}$ and edges: $A \to B$, $B \to A$ and $A \to C$.
Start DFS from vertex A, then visit B we have $B.d = 2, B.f = 3$. After finishing exploring B, start DFS from C and $C.d = 4$ and $C.f = 5$. So the path $B \to C$ exists, and $B.d(2) < (4)C.d$, but C is **not** a descendant of B in the DFS forest because the DFF forest produced by DFS is:



(b) If a directed graph G contains a path from u to v, then any depth-first search of G must result in $v.d \leq u.f$.

Answer

Similarly consider the directed graph G with vertices $\{A, B, C\}$ and edges: $A \to B$, $B \to A$ and $A \to C$ shown above. Start DFS from vertex A, then visit B we have $B.d = 2, B.f = 3$. After finishing exploring B, start DFS from C and $C.d = 4$ and $C.f = 5$. We see that $B.f = 3$ and $C.d = 4$. Although there is a path from B to C, we have $C.d(4) > B.f(3)$ hence the contrary of the statement is true.

5. **The knight**
A generalized chess board has $N \times N$ squares for some $N \geq 1$. Each square is identified by its two dimensional coordinates (i, j), where $i, j \in [1, N]$. A knight placed at a location (i, j) makes a move by moving exactly two steps in some cardinal direction and one step in an orthonogoal direction, so long as the new location is on the board—just as a normal knight does in the game of chess. Design an algorithm that given a board dimension N , an initial position of the knight (i, j), and a final position of the knight (i', j'), calculates as output the minimum number of moves the knight needs to make to reach the final position from the initial position. (The minimum is ∞ if the final position is not reachable by a knight making moves starting at the initial position.) Design the algorithm to have as low a time complexity as possible. Along with the algorithm, state and explain its time complexity.

Answer

Algorithm knight problem

```
Function knight(C[1...N, 1...N], start, end):
    if start = end then
        return 0
    moves ← [(-2, -1), (-1, -2), (1, -2), (2, -1), (2, 1), (1, 2), (-1, 2), (-2, 1)]

    visited ← 2D array of size N × N initialized to false

    queue ← empty queue
    enqueue (start, 0) into queue
    mark start as visited
    while queue is not empty do
        (curr, dist) ← dequeue from queue
        foreach move in moves do
            next ← curr + move
            if next is within bounds and not visited then
                if next = end then
                    return dist + 1
                mark next as visited
                enqueue (next, dist + 1)
    return ∞
```

Explanation: The algorithm performs a **breadth-first search (BFS)** starting from the initial position of the knight. At each level of the BFS, it explores all valid knight moves from the current square. The knight has exactly 8 possible moves, and we enqueue each reachable, unvisited position with its corresponding distance from the start. The BFS guarantees that the first time we reach the target square, we have found the shortest path to it (i.e., the minimum number of moves).
To ensure we do not visit the same position more than once, we maintain a 2D `visited` array. We stop the search as soon as the destination is reached, or we exhaust all reachable squares (in which case the destination is unreachable).
Time Complexity: Each cell on the $N \times N$ board can be visited at most once, and for each cell, we consider 8 constant-time knight moves. Thus, the time complexity is:

$$O(N^2)$$

which is optimal for this type of grid-based shortest path search.
Space Complexity: The space used is also $O(N^2)$ for the `visited` array and the BFS queue.

6. **A Different Implementation of Dijkstra's**

   (a) Let $G = (V, E)$ be a weighted, directed graph with nonnegative weight function $w : E \to \{0, 1, 2, \ldots W\}$, for some nonnegative integer W. Implement Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time. Explain why your algorithm is correct and why its time complexity to be $O(WV + E)$.

Answer

Algorithm Dial's Algorithm

Function `DialDijkstra`$(G = (V, E), s, W)$**:**

```
Initialize dist[v] ← ∞ for all v ∈ V − {s}, dist[s] ← 0
Initialize buckets[0 ... W · |V|] as empty lists
Insert s into buckets[0]
curr_index ← 0

while curr_index < W · |V| do
    while buckets[curr_index] is empty do
        curr_index ← curr_index + 1
        if curr_index = W · |V| then
            return dist
    Remove vertex u from buckets[curr_index]

    foreach edge (u, v) ∈ E with weight w(u, v) do
        if dist[v] > dist[u] + w(u, v) then
            old_dist ← dist[v]
            dist[v] ← dist[u] + w(u, v)
            Insert v into buckets[dist[v]]
return dist
```

Explanation: The algorithm uses an array of buckets indexed by *tentative distance values*. Each bucket at index i holds all vertices whose current shortest-path estimate is i. This structure ensures that vertices are processed in non-decreasing order of distance—just like in Dijkstra's algorithm with priority queue.

Since all edge weights are integers in $\{0, 1, \ldots, W\}$, the maximum possible shortest-path distance from the source to any vertex is at most $W \cdot |V|$. Therefore, we only need $W \cdot |V|$ buckets.
Each vertex may be updated and reinserted into a new bucket at most W times (once per possible improvement via edge weights), and each edge is relaxed at most once. Bucket operations (insertion and deletion) are constant-time, but finding the next non-empty bucket may take linear time in the worst case. Over the entire run, the cost of scanning buckets is $O(WV)$, and edge relaxations take $O(E)$. Thus, the total runtime is $O(WV + E)$.

The algorithm is correct because it maintains the same invariant as Dijkstra's algorithm: vertices are processed in order of increasing tentative distance, and once a vertex is processed, its distance is finalized due to nonnegative weights. While the priority queue version achieves logarithmic time for `extract-min`, this bucket-based implementation is more efficient when edge weights are small.