Theory Assignment-5: ADA Winter-2024

Vikranth Udandarao (2022570) Ansh Varshney (2022083)

1 Preprocessing and Assumptions

In this problem, we preprocess, assume or have been given the following:

1. All the boxes have dimensions strictly between 10 cm and 20 cm in height, width, and depth.
2. Boxes can be nested if one box can be rotated to fit inside another, considering height, width, and depth constraints.

2 Algorithm Description

This section elaborates on the adaptation of the Ford-Fulkerson method to solve the problem of minimizing the number of visible boxes by optimally nesting them, conceptualizing the problem using a bipartite graph model.

2.1 Formulating as a Flow-Network Problem

To address the box nesting problem, we first represent each box as a vertex in a flow network, creating a bipartite graph where each side of the bipartition represents a set of boxes. One side can potentially be nested inside boxes of the other side based on their dimensions. Additional vertices include a source s and a sink t, completing the flow network.

2.2 Bipartite Flow Network Construction

1. **Vertices Representation:**
   - Each box is treated as a vertex. The bipartite nature segregates boxes into two disjoint sets where edges only connect vertices from opposite sets, representing feasible nesting.
2. **Edges Creation:**
   - Directed edges are established between vertices across the bipartite sets if a box from one set can be nested inside a box from the opposite set.
   - The source vertex s is connected with edges to all boxes that potentially contain others, and all boxes that can be contained connect to the sink vertex t.
3. **Capacities Assignment:**
   - As typical in bipartite graphs used in flow networks, the capacity of each edge is set to 1, representing the single opportunity for one box to nest into another.

2.3 Applying the Ford-Fulkerson Algorithm in Bipartite Context

Using the Ford-Fulkerson method, the algorithm seeks to maximize the flow from s to t, reflecting the optimal nesting of boxes. Each iteration of the algorithm enhances the flow by finding augmenting paths in the bipartite graph, effectively calculating the maximum matching between the sets of boxes that can nest one another.

The maximum flow found corresponds directly to the maximum number of boxes that can be nested within each other, minimizing the number of boxes that remain visible, i.e., those that do not participate in any matching and thus are not nested.

This approach not only provides an efficient computational method but also leverages the structural properties of bipartite graphs to ensure an optimal and correct solution to the nesting problem.

3 Pseudocode

4 Complexity Analysis

4.1 Time Complexity

1. **Constructing the Flow Network:**
   - Constructing vertices and edges in the network involves examining each pair of boxes to determine if one can nest inside another. For n boxes, this results in $O(n^2)$ comparisons.
2. **Ford-Fulkerson Algorithm:**
   - The Ford-Fulkerson algorithm involves finding augmenting paths and updating the flow until no augmenting paths remain. The worst-case time complexity is $O(max\_flow \times (|V| + |E|))$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.
   - In our scenario, since each box can only be nested once and the capacity of each edge is 1, the maximum flow is at most n. Therefore, the complexity becomes $O(n \times (n + n^2))$, simplifying to $O(n^3)$.

Combining these components, the overall time complexity of the algorithm can be considered as $O(n^3)$, driven primarily by the Ford-Fulkerson component.

4.2 Space Complexity

1. **Flow Network Storage:**
   - The adjacency matrix used to represent the capacity between each pair of vertices requires $O(V^2)$ space, where V is the number of vertices, including the source and sink.
2. **Auxiliary Space:**
   - Additional space is required for maintaining parent arrays, queue data structures during BFS, and other temporary data during flow updates. This adds up to $O(V)$ space.

Thus, the total space complexity of the algorithm is $O(V^2)$, where V includes all box vertices plus the source and sink nodes, effectively resulting in $O((n + 2)^2)$ which simplifies to $O(n^2)$.

5 Necessary and Sufficient Conditions explaining the algorithm

The correctness of the Ford-Fulkerson algorithm in solving the box nesting problem hinges on several necessary and sufficient conditions related to the properties of the flow network and the nature of the flow itself. These conditions ensure that the algorithm not only terminates but also yields the optimal solution—minimizing the number of visible boxes.

5.1 Necessary Conditions

These are conditions that must be met for the algorithm to function correctly:

1. **Network Representation Validity:** The flow network must accurately represent all potential nesting relationships among the boxes. This involves correct vertex creation for each box and appropriate directed edges that reflect possible nesting, based on the dimensional comparisons of the boxes.
2. **Proper Initialization:** The initial flow in the network must be zero, and capacities on all edges must be set correctly—edges between boxes should have a capacity of 1 (indicating a single nesting opportunity), and similarly, edges from the source to boxes and from boxes to the sink must also have a capacity of 1 to ensure that each box is considered only once.
3. **Path Augmentation Integrity:** During each iteration, the path found by the algorithm (using BFS or DFS) that increases the flow must be an augmenting path with available capacity. This ensures that each flow increment is feasible given the current state of the network.

Algorithm 1 Minimizing Visible Boxes

```
function canNestInside(Box a, Box b)
    return (a.height < b.height) and (a.width < b.width) and (a.depth < b.depth)    ▷ Check if Box a can nest inside Box b
end function

function addEdge(vector<vector<int>> &capacity, int from, int to)
    capacity[from][to] ← 1                                                          ▷ Create directed edge with capacity 1
end function

function fordFulkerson(vector<vector<int>> capacity, int source, int sink)
    int totalFlow ← 0                                                               ▷ Initialize total flow to 0
    int numVertices ← capacity.size()                                               ▷ Get the number of vertices in the graph
    <vector> residualCapacity ← capacity                                            ▷ Create a copy of capacity for residual calculations
    while true do                                                                   ▷ Continue until no augmenting path is found
        vector<int> parent(numVertices, -1)                                         ▷ Track the parent of each vertex
        queue<int> q                                                                ▷ Queue for BFS
        q.push(source)
        parent[source] ← source                                                     ▷ Start BFS from the source
        while not q.empty() and parent[sink] == -1 do                               ▷ Execute BFS until the sink is reached
            int u ← q.front()
            q.pop()
            for int v = 0 to numVertices - 1 do
                if parent[v] == -1 and residualCapacity[u][v] < 0 then
                    parent[v] ← u
                    q.push(v)
                    if v == sink then
                        break                                                       ▷ Break if sink is reached
                    end if
                end if
            end for
        end while
        if parent[sink] == -1 then
            break                                                                   ▷ No augmenting path, so break the loop
        end if
        int pathFlow ← INT_MAX                                                      ▷ Find the minimum capacity in the path from source to sink
        for int v = sink; v != source; v = parent[v] do
            int u ← parent[v]
            pathFlow ← min(pathFlow, residualCapacity[u][v])
        end for
        for int v = sink; v != source; v = parent[v] do                             ▷ Update the residual capacities of the path edges
            int u ← parent[v]
            residualCapacity[u][v] -= pathFlow
            residualCapacity[v][u] += pathFlow
        end for
        totalFlow += pathFlow                                                       ▷ Add path flow to total flow
    end while
    return totalFlow                                                                ▷ Return the total flow, which is the max flow
end function
```

5.2 Sufficient Conditions

These are conditions that, when satisfied, guarantee the optimality and correctness of the algorithm's outcome:

1. **Conservation of Flow:** At every non-source, non-sink vertex in the network, the total flow into a vertex must equal the total flow out of it. This flow conservation principle is crucial for ensuring that no flow is lost or created within the network, mirroring the physical reality that each box can either nest another or be nested.

2. **Absence of Augmenting Paths at Termination:** The Ford-Fulkerson algorithm terminates when no more augmenting paths from the source to the sink can be found in the residual graph. The absence of such paths means that it is not possible to push additional flow through the network, indicating that the maximum flow has been achieved.

3. **Max-Flow Min-Cut Theorem:** The maximum flow through the network equals the capacity of the minimum cut in the network (the smallest set of edges that, if removed, would disconnect the source from the sink). This theorem not only provides a stopping criterion but also assures us that the maximum number of boxes have been nested optimally, minimizing the number of visible boxes.

These conditions collectively guarantee that the Ford-Fulkerson method correctly identifies the maximum possible nesting of boxes, which directly translates to the minimum number of visible boxes. By ensuring that these conditions are met, we can confidently claim that the solution provided by the algorithm is not only feasible but also optimal.

6 Proof of Correctness

This section provides a formal proof of the correctness of the Ford-Fulkerson algorithm when applied to the problem of minimizing the number of visible boxes by maximizing the nesting of boxes within each other, considering the bipartite nature of the constructed flow network.

6.1 Basis of the Proof

The proof is based on the following pillars:

1. The flow network correctly represents the nesting possibilities between boxes as a bipartite graph.

2. The Ford-Fulkerson algorithm finds the maximum flow in this network.

3. The maximum flow corresponds to the maximum number of boxes that can be nested within each other.

4. The number of boxes minus the maximum flow yields the minimum number of visible boxes.

6.2 Step-by-Step Proof

1. **Flow Network Validity:**

   - Each box is represented as a vertex. An edge from vertex i to vertex j exists if box i can be nested inside box j, based on their dimensions, forming a bipartite graph where vertices are divided into two disjoint sets based on the possibility of nesting.
   - Source and sink vertices are added, with source edges directed to every box and sink edges directed from every box, all with unit capacity. This setup ensures that each box's use in nesting is accounted for exactly once.

2. **Bipartite Graph Structure:**

   - The bipartite graph effectively partitions the boxes into two sets where nesting relationships are possible only between elements of different sets, enhancing the clarity and efficiency of the network's representation.

- This structure simplifies the search for augmenting paths, as each path alternates between the sets, which is characteristic of bipartite matching.

3. **Application of Ford-Fulkerson:**

   - The algorithm repeatedly finds augmenting paths from the source to the sink with available capacity. For each path, it adjusts the flow to push the maximum amount possible until no further augmenting paths exist.
   - By the properties of flow conservation and the nature of the Ford-Fulkerson method, each augmentation step ensures that the flow value is incremented optimally, reflecting an actual increase in nesting levels where possible.

4. **Correspondence to Maximum Nesting:**

   - When the algorithm terminates, the maximum flow through the network has been achieved. This flow value, by construction of the network, represents the maximum number of boxes that can be nested inside each other.
   - The termination occurs when no more augmenting paths can be found, which by the Max-Flow Min-Cut Theorem corresponds to the situation where the minimum cut of the network equals the flow value—signifying that no additional nesting is feasible without exceeding the physical constraints of the boxes.

5. **Calculating Visible Boxes:**

   - Since each unit of flow represents a box being nested within another, the total number of boxes minus the maximum flow gives the number of boxes that cannot be nested any further, thus remaining visible.

6. **Conclusion:**

   - The maximum flow thus directly correlates with the minimum number of visible boxes. Since the flow maximizes the internal nesting of boxes, reducing the external visibility count, the solution is both optimal and correct given the problem constraints.

This structured proof, built on fundamental principles of flow networks and optimization algorithms, rigorously demonstrates that the Ford-Fulkerson method, as applied here, yields an accurate and optimal solution to minimizing the number of visible boxes through nesting. The inclusion of the bipartite graph model further solidifies the theoretical correctness of the algorithm by leveraging its properties for more efficient matching and flow calculation.

Algorithm 2 Main

```
1: function MAIN
2:     int n ← 5                                                          ▷ Example testcase
3:     vector<Box> boxes ← {Box(11, 12, 13), Box(15, 16, 17), Box(14, 15, 16), Box(18, 19, 20), Box(12, 13,
   14)}                                                                   ▷ Predefined box dimensions
4:     int source ← n                                                     ▷ Define source vertex index
5:     int sink ← n + 1                                                   ▷ Define sink vertex index
6:     vector<vector<int>> capacity(n + 2, vector<int>(n + 2, 0))         ▷ Initialize the capacity matrix
7:     for int i = 0 to n - 1 do
8:         for int j = 0 to n - 1 do
9:             if i != j and canNestInside(boxes[i], boxes[j]) then
10:                addEdge(capacity, i, j)                                ▷ Add edge if one box can nest inside another
11:            end if
12:        end for
13:    end for
14:    for int i = 0 to n - 1 do
15:        addEdge(capacity, source, i)                                   ▷ Connect source to all boxes
16:        addEdge(capacity, i, sink)                                     ▷ Connect all boxes to sink
17:    end for
18:    int minVisibleBoxes ← n - FORDFULKERSON(capacity, source, sink)    ▷ Calculate minimum visible boxes
19:    cout ¡¡ "Minimum number of visible boxes: " ¡¡ minVisibleBoxes      ▷ Output the result
20: end function
```