Theory Assignment-5: ADA Winter-2024

Aditya Sharma (2022038) Ayan Kumar Singh (2022122)

1 Algorithm Description

The algorithm uses the concept of bipartite matching to find the maximum number of nested boxes. The bipartite graph is constructed by considering each box as a vertex and adding an edge between two vertices if one box can contain the other. The Augmenting Path algorithm is then used to find the maximum bipartite matching, which corresponds to the maximum number of nested boxes.

2 Formulating the Problem as a Flow Network

To formulate the problem as a flow network problem, we represent it as a bipartite graph where each node corresponds to a box, and edges represent the possibility of one box being nested inside another. We get a source node s and a sink node t. Each edge from s to a box node will have a capacity of 1, representing the possibility of nesting that box. Also, each edge from a box node to t will also have a capacity of 1, showing that the box can be nested inside another.

2.1 Construction of the Flow Network

- We define a bipartite graph G as follows:
- **Vertices**: G has $2n$ vertices $u_1, u_2, \ldots, u_n, v_1, v_2, \ldots, v_n$. Vertices u_i and v_i correspond to box i, represented as nodes in sets U and V respectively. Additionally, we have a source node s and a sink node t.
- **Edges**: G contains an edge (u_i, v_j) if and only if box i can be placed inside box j.
  - For each box node in set U, we add an edge from the source s with capacity 1.
  - For each box node in set V, we add an edge to the sink t with capacity 1.
  - For a pair of nodes (u, v) where u is a box node in U and v is a box node in V, if u can contain v following given conditions, we add an edge from u to v with capacity 1.
- **Capacity Assignment**: Each edge from s to a box node and from a box node to t has a capacity of 1, indicating that only one box can be nested inside another.

3 Justification for minimum cut size

- Consider a valid nesting of boxes where box i is directly nested inside box j if i is inside j without any intervening boxes.

- In this nesting arrangement, each box is either visible (not nested inside any other box) or nested directly inside exactly one other box.
- The constraints on box dimensions ensure that at most one box can be directly nested inside another.
- Any legal nesting of the boxes can be represented as a matching in graph G, where an edge $u_i v_j$ exists in the matching if box i is directly inside box j.
- Conversely, any matching M in G assigns each box to lie directly inside at most one other box. For any edge $u_i v_j$ in G, the smallest dimension of box i is strictly smaller than the smallest dimension of box j.
- Thus, the arrangement induced by matching M cannot have a cycle of nested boxes, ensuring that M corresponds to a valid nesting.
- In any valid nesting, a box i is visible if and only if no edge in the corresponding matching involves u_i.
- Therefore, the number of visible boxes is exactly n minus the number of edges in the matching.
- Consequently, the nesting arrangement with the fewest visible boxes corresponds to the maximum-cardinality matching in graph G.

To explain why the maximum flow value or minimum cut size corresponds to the actual answer to the problem of nesting boxes to minimize visible boxes, we relate this problem to the concept of flow networks. Below are the steps that are followed:

- **Graph Representation**: Model the problem as a directed graph where each box is a node. Include special source (s) and sink (t) nodes.
- **Edge Capacities**: Assign capacities to edges based on nesting feasibility between boxes.
- **Maximum Flow**: Maximize flow from s to t, representing maximum nesting of boxes.
- **Minimum Cut**: The capacity of the minimum s-t cut in the flow network equals the number of visible boxes (those not nested inside others).
- **Relation to Problem**:
  - Maximum flow value = Maximum number of nested boxes.
  - Minimum cut size = Minimum number of visible boxes.

The maximum flow value obtained from the Ford-Fulkerson algorithm corresponds to the maximum number of nested boxes. Each unit of flow represents nesting one box inside another, subject to the capacity constraints of the edges. Therefore, the maximum flow value directly corresponds to the maximum number of nested boxes.

4 Complexity Analysis

4.1 Time Complexity

- **Construction of Flow Network**: $O(n^2)$ time, where n is the number of boxes.
- **Ford-Fulkerson Algorithm**: The time complexity depends on the number of augmenting paths found during the process. It's bounded by $O(\text{value(flow)} \times (n^2))$ time.
- In this case, the input size is the number of boxes 'n'.
- The time complexity of the algorithm is dominated by the bipartite matching algorithm, which uses the Augmenting Path algorithm. The time complexity of the Augmenting Path algorithm is $O(V * E)$, where V is the number of vertices, and E is the number of edges in the bipartite graph.
  In this problem:
  - V = n (the number of boxes)
  - E = $O(n^2)$ in the worst case, when all boxes can contain each other

  Therefore, the time complexity of the bipartite matching algorithm is

$$O(n * n^2) = O(n^3)$$

4.2 Space Complexity Analysis:

In this case, the input size is the number of boxes 'n'.

1. **'boxes' 2D vector:** This vector stores the dimensions of each box. It has a size of 'n' rows and 3 columns (for width, height, and depth). Therefore, the space required for 'boxes' is O(n).
2. **'graph' adjacency list:** This is a vector of vectors, where each inner vector represents the neighbors of a vertex (box) in the bipartite graph. In the worst case, when all boxes can contain each other, the graph can have $O(n^2)$ edges. Therefore, the space required for 'graph' is $O(n^2)$.
3. **'match' array:** This array stores the matching information for each vertex in the bipartite graph. It has a size of 'n'. Therefore, the space required for 'match' is $O(n)$.
4. **'visited' array:** This array is used to keep track of visited vertices during the bipartite matching algorithm. It has a size of 'n'. Therefore, the space required for 'visited' is $O(n)$.

Putting it all together, the total space complexity of the algorithm is:

$$O(n) + O(n^2) + O(n) + O(n) = O(n^2)$$

Therefore, the space complexity of the algorithm is $O(n^2)$, where 'n' is the number of boxes.

5 Algorithm Steps

The algorithm implemented in the given code follows these steps:

1. Read the number of boxes ‘n‘ from the input.
2. Read the dimensions (width, height, depth) of each box and store them in the ‘boxes‘ 2D vector.
3. Construct the bipartite graph by iterating over all pairs of boxes: For each pair of boxes ‘i‘ and ‘j‘, if ‘i‘ is not equal to ‘j‘ and ‘canContain(i, j)‘ returns ‘true‘ (i.e., box ‘i‘ can contain box ‘j‘), add an edge from ‘i‘ to ‘j‘ in the ‘graph‘ adjacency list.
4. Call the ‘maxMatching‘ function to find the maximum bipartite matching:
   (a) Initialize the ‘match‘ array with -1 (indicating no match).
   (b) For each unmatched vertex ‘i‘ in the graph:
      i. Reset the ‘visited‘ array to ‘false‘.
      ii. Call the ‘bpm‘ (Bipartite Perfect Matching) function with ‘i‘ as the starting vertex.
      iii. If ‘bpm‘ returns ‘true‘, increment the ‘maxMatch‘ counter.
   (c) Return the ‘maxMatch‘ value, which represents the maximum number of nested boxes.
5. Calculate the number of visible boxes as ‘n - maxNestedBoxes‘.
6. Print the maximum number of nested boxes and the number of visible boxes.

5.1 ‘bpm‘ function

The ‘bpm‘ function implements the Augmenting Path algorithm for finding a maximum bipartite matching:

1. For each unvisited neighbor ‘v‘ of the current vertex ‘u‘:
   (a) Mark ‘v‘ as visited.
   (b) If ‘v‘ is unmatched (‘match[v] == -1‘) or there exists an augmenting path starting from ‘match[v]‘:
      i. Match ‘v‘ with ‘u‘ by setting ‘match[v] = u‘.
      ii. Return ‘true‘ to indicate that an augmenting path has been found.
2. If no augmenting path is found, return ‘false‘.

5.2 ‘canContain‘ function

The ‘canContain‘ function checks if one box can contain another by comparing their dimensions:

1. For each dimension ‘i‘ of box1:
   (a) Initialize ‘canFit‘ to ‘true‘.
   (b) For each dimension ‘j‘ of box2:

i. If ‘boxes[box1][j] >= boxes[box2][(i+j)%3]‘ (i.e., box1’s ‘j‘-th dimension is greater than or equal to box2’s rotated ‘(i+j)%3‘-th dimension), set ‘canFit‘ to ‘false‘ and break the inner loop.

(c) If ‘canFit‘ remains ‘true‘ after checking all dimensions of box2, it means that box1 can contain box2. Return ‘true‘.

2. If no dimension of box1 can contain the corresponding rotated dimensions of box2, return ‘false‘.

6 PseudoCode

Algorithm 1 Maximum Nested Boxes Algorithm

```
procedure MAIN
    Read the number of boxes n
    Read the dimensions (width, height, depth) of each box and store in boxes
    Construct the bipartite graph G = (U, V, E) by calling CONSTRUCTGRAPH(n)
    maxNestedBoxes ← MAXMATCHING(n)
    visibleBoxes ← n − maxNestedBoxes
    Print maxNestedBoxes and visibleBoxes
end procedure
procedure CONSTRUCTGRAPH(n)
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            if i ≠ j and CANCONTAIN(i, j) then
                Add an edge from i to j in G
            end if
        end for
    end for
end procedure
procedure CANCONTAIN(box1, box2)
    for i ← 0 to 2 do
        canFit ← True
        for j ← 0 to 2 do
            if boxes[box1][j] ≥ boxes[box2][(i + j) mod 3] then
                canFit ← False
                break
            end if
        end for
        if canFit then
            return True
        end if
    end for
    return False
end procedure
procedure MAXMATCHING(n)
    Initialize match array with −1
    maxMatch ← 0
    for i ← 0 to n − 1 do
        Initialize visited array with False
        if BPM(i) then
            maxMatch ← maxMatch + 1
        end if
    end for
    return maxMatch
end procedure
procedure BPM(u)
    for v in neighbors of u in G do
        if visited[v] = False then
            visited[v] ← True
            if match[v] = −1 or BPM(match[v]) then
                match[v] ← u
                return True
            end if
        end if
    end for
    return False
end procedure
```

7 Main Function

- This is the main function that we use to drive our *Maximum Nested Boxes Algorithm*.

```cpp
int main() {
    int n; // Number of boxes
    cin >> n;

    // Read box dimensions (width, height, depth) for each box
    boxes.resize(n, vector<int>(3));
    for (int i = 0; i < n; ++i) {
        cin >> boxes[i][0] >> boxes[i][1] >> boxes[i][2];
    }

    // Constructing the bipartite graph
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i != j && canContain(i, j)) {
                graph[i].push_back(j);
            }
        }
    }

    int maxNestedBoxes = maxMatching(n);
    int visibleBoxes = n - maxNestedBoxes;

    cout << "Maximum number of nested boxes: " << maxNestedBoxes << endl;
    cout << "Number of visible boxes: " << visibleBoxes << endl;

    return 0;
}
```

Listing 1: C++ code snippet of main function