Homework 2

Gleici Pereira, Rachel Collier

March 2023

1

Consider the set cover problem on U with n elements. Explain for which inputs the greedy algorithm can be $O(n)$ and for which inputs it can be $O(2^n)$. How far is the greedy algorithm solution from the optimal solution?

1.1 Solution

The set cover problem is a NP-hard optimization problem which finds the minimum subset of a subset S whose union equals U.

Because of its NP-hardness nature, the best way to approach this problem is with a greedy algorithm that gives a logarithmic time complexity approximation, which is the best possible.

1.2 Algorithm

Algorithm 1 Greedy Set Cover
Input: A universal set U and subsets S
Output: A collection of subsets C that cover U

```
1: function greedyCover(U, S)
2:     C ← ∅
3:     while C ≠ U do
4:         S_i ← subset which covers the most elements of U
5:         C = C ∪ S_i
6:     end while
7:     return C
8: end function
```

1.3 Analysis

This greedy algorithm gives a O(logΔ) approximation. That is, there are certain inputs that might result in better or worse running times. In particular, it

can yield O(n) running time when the subsets S of U have a small size resulting in a small number of subsets evaluations which means it would approach linear time complexity. However, if the number of subsets S is large enough then that means the algorithm needs to do more evaluations resulting in a slower running time approaching exponential time O(2^n).

Considering the differences in running times according to the inputs given, it is reasonable to state that in some cases the greedy algorithm will produce outputs far from the optimal solution, which is defined as the solution that has covered all the elements of U. However, it is proven that the approximation ratio of the greedy algorithm is O(log(n/t)) = O(logΔ), where Δ is the maximum number of elements covered by any subset. Therefore, the greedy algorithm for the cover set problem is not far from the optimal solution in some cases. Indeed, depending on the number of subsets, size of U, and structure of problem it is very close to the optimal solution.

2

Assume you have a string of length n with k symbols, possibly repeated. Justify why Huffman codes can be built in time $\Theta(nlog(n))$. Explain in which cases RLE is better than Huffman codes. Show the algorithms and their $O()$ to encode a string with n symbols and decode a string with m bits.

2.1 Solution

Huffman coding is a compression algorithm that assigns variable-length codes to symbols based on their frequencies in each input string. The algorithm constructs a binary tree of nodes that represent symbols and their frequencies, where the path to a symbol in the tree corresponds to its Huffman code.

The construction of the Huffman binary tree involves counting the frequency of each symbol in the input string. This can be done in O(n) time, where n is the length of the input string. Build a priority queue of nodes, one for each symbol, with the frequency of the symbol as the priority. This can be done in O(k log k) time, where k is the number of distinct symbols in the input string. Repeat the following steps until only one node remains in the priority queue. The above steps can be done in O(k log k) time, where k is the number of distinct symbols in the input string. Therefore, the total time complexity of building the Huffman binary tree is O(n + k log k). Since k is at most n, the time complexity can be simplified to O(n log n), which is the upper bound for the time complexity of constructing the Huffman binary tree. In conclusion, the time complexity of building the Huffman codes for a string of length n with k symbols is (n log(n)).

Run-length encoding (RLE) is a simple compression algorithm that replaces sequences of repeated symbols in a string with a count of the number of times

that symbol is repeated. In some cases, RLE can produce a shorter encoded string than Huffman codes. RLE is particularly effective for strings with long runs of repeated symbols, such as images or sound recordings. For example, a black-and-white image with a large area of solid white could be encoded with RLE as a single white pixel followed by a count of the number of white pixels in the run, resulting in a much shorter encoded string than Huffman codes.

The encoding algorithm is to initialize a count variable to 1 and a current symbol variable to the first symbol in the input string. For each subsequent symbol in the input string, the symbol is the same as the current symbol, increment the count variable. If the symbol is different from the current symbol, output the current symbol and its count, and set the count variable to 1 and the current symbol variable to the new symbol. The output the final symbol and its count is the time complexity of the RLE encoding algorithm is O(n), where n is the length of the input string.

3

For the DNA sequence alignment problem show the top-down algorithm with smart recursion. Explain how this algorithm is transformed into the classic "dynamic programming" using loops. Explain why it is O(mn) for input strings of length m, n. Compare this algorithm with the EditDistance algorithm in Jeff's textbook.

3.1 Solution

The top-down algorithm presented for the DNA sequence alignment problem is a recursive algorithm that uses memoization to avoid recomputing the same subproblems multiple times. However, it can be transformed into a dynamic programming iterative algorithm that computes each subproblem only once, using loops to fill in a memoization table.

To convert the top-down algorithm into an iterative algorithm, we first initialize a two-dimensional memoization table of size (m+1) x (n+1), where m and n are the lengths of the two input sequences. We then fill in the table using a nested loop, with the outer loop iterating over i from 0 to m, and the inner loop iterating over j from 0 to n. For each i, j pair, we check whether the value of the corresponding memoization table entry has already been computed. If not, we compute it using the recurrence relation from the top-down algorithm, i.e., we consider the three possible operations (insertion, deletion, or substitution) and their associated costs, and recursively compute the minimum cost. The time complexity of this dynamic programming iterative algorithm is O(mn). This is because the algorithm fills in a table of size (m+1) x (n+1), and for each table entry, it considers at most three possible options. Therefore, the total number of operations performed by the algorithm is proportional to the number of entries

in the table, which is (m+1)(n+1). Hence, the time complexity is O(mn) for input strings of length m and n.

Here are some similarities between the top-down algorithm and the EditDistance algorithm. Both algorithms involve filling a matrix with values to determine the optimal alignment. Both algorithms have a time complexity of O(mn), where m and n are the lengths of the two input sequences.

Here are some differences between the top-down algorithm and the EditDistance algorithm. The top down algorithm works recursively and constructs the optimal alignment from the top down, while EditDistance works iteratively and constructs the optimal alignment from the bottom up. The top down algorithm computes the cost of all possible alignments between two subsequences, while EditDistance computes the minimum cost of all possible ways to transform one sequence into another by a set of edit operations.

4

Write a table summarizing $O()$ for these operations: insert, delete, search one element, get max(), list all elements in order of these data structures: sorted linked list, unsorted array, sorted array stack, circular queue, binary search tree (balanced), heap. If an operation does not make sense write "NA" (not available).

Data Structure	Insert	Delete	Search	Get Max
Sorted Linked List	O(n)	O(n)	O(n)	O(1)
Unsorted Array	O(1)	O(n)	O(n)	O(n)
Sorted Array (Stack)	O(n)	O(1)	O(log n)	O(1)
Circular Queue	O(1)	O(1)	O(n)	NA
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)
Balanced Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)
Heap (max)	O(log n)	O(log n)	NA	O(1)

5

For the matrix-chain multiplication problem compare the smart recursion (DP) solution with these alternative solutions: (1) exhaustive: consider all potential parenthesis placements (2) greedy: pick the product with the largest intermediate sizes. (3) greedy: pick the product with the smallest matrix output size. Write the algorithm for each alternative and show its $O()$.

5.1 Solution

The matrix-chain multiplication analyzes the running times of matrix multiplication by considering we can compute the product of matrices in different ways

and that depending on how we parenthesize matrices lead to different running times and costs. Its goal is to find an optimal solution that minimizes the cost of multiplying M matrices.

The smart recursion solution constructs a solution by breaking the problem into sub-problems and computing the solution without redoing computations.

5.2 Algorithm Smart Solution

Algorithm 2 Smart Recursion Algorithm

Input: n matrices with dimensions p

```
1:  function SmartSol(p)
2:      for i = 1 to n do
3:          m[i,j] = 0
4:          for l = 2 to n do
5:              for i = 1 to n-l+1 do
6:                  j = i+l-1
7:                  m[i, j] = ∞
8:                  for k = i to j-1 do
9:                      q = m[i, k] + m(k + 1, j) + p_{i-1} p_k p_j
10:                     if q < m[i, j] then
11:                         m[i, j] = q
12:                         return m[i, j]
13:
```

(1) The exhaustive solution considers all potential parenthesis orders and calculates the number of multiplications for each arrangement. The min value starts at ∞ and is updated every time a smaller value is found. The function returns the min cost at the end of the algorithm.

5.3 Exhaustive Solution Algorithm

Algorithm 3 Exhaustive Solution
Input: An array A with n elements
Output: Minimum cost for

```
1: function EXHAUSTIVESOL(A, i, j)
2:     if i == j then
3:         return 0
4:     end if
5:     minCost = ∞
6:     for k ← 1 to j − 1 do
7:         c = exhaustiveSol(A, i, k) + exhaustiveSol(A, k+1, j) + p_{i−1} * p_k * p_j
8:         if c < minCost then
9:             minCost ← cost
10:        end if
11:    end for
12: end function=0
```

The exhaustive solution has $\theta(2^n)$ time complexity because there are 2^{n-1} possible ways to arrange n matrices. Thus, this algorithm is exponential.
(2) The first greedy solution picks the product with the largest intermediate size at each iteration and multiplies it with the remaining matrices.

6

Consider two large sets of integers S_1 and S_2 and a limited amount of RAM. Write efficient algorithms to compute $S_1 \cap S_2$, $S_1 \cup S_2$, $S_1 \subseteq S_2$, assuming only a block (subset) of each set can be loaded in RAM. What is the fastest algorithm you can come up with? Show its $O()$.

6.1 Solution

The fastest algorithm to solve this problem that I can think of is the simple external Merge-Sort, which is a modified version of the classic Merge-Sort algorithm and still runs in $\theta(log(n))$. It works by dividing the sets in half and processing the computations in chunks.

External Mergesort Algorithm
1. Split sets S_1 and S_2 into two blocks of size B.
2. Sort each block using quicksort algorithm
3. Write the sorted blocks to disk
4. Compute any operations on the blocks
4. Finally, merge the blocks with the merge routine

$S_1 \cup S_2$ Algorithm
1. Split sets S_1 and S_2 into two blocks of size B.
2. Sort each block using quicksort algorithm
3. Write the sorted blocks to disk
4. From element i = 1 to element n 5. Compare the elements of both blocks
6. If elements are equal, output any element and increment the index of both blocks
7. Else if the first element is less than the second element, output it and increment the index of first block.
8. Else if the second element is less then the first element, then output it and increment the index of second block.
9. Merge blocks with the the merge routine.

$S_1 \cap S_2$ Algorithm
1. Split sets S_1 and S_2 into two blocks of size B.
2. Sort each block using quicksort algorithm
3. Write the sorted blocks to disk
4. From element i = 1 to element n 5. Compare the elements of both blocks
6. If elements are equal, output the element and increment the index of both blocks
7. Else if the first element is less than the second element, increment the index of first block.
8. Else if the second element is less then the first element, increment the index of second block. 9. Merge blocks with the the merge routine.

$S_1 \subseteq S_2$ Algorithm
1. Split sets S_1 and S_2 into two blocks of size B.
2. Sort each block using quicksort algorithm
3. Write the sorted blocks to disk
4. Compute the intersection of sets
5. If the size of the intersection equals the size of S_1, then S_1 is a subset of S_2.
6. Return S_1

7

Contrast Kruskal and Prim algorithms to find the MST of a connected graph G. Compare the $O()$. Now, consider the dynamic case where edges or vertices may be inserted or deleted continuously. Which algorithm is better to handle a batch of changes?

7.1 Solution

Kruskal's algorithm: Sort all the edges in non-decreasing order of their weight. Pick the smallest edge. If adding the edge creates a cycle, discard it. Otherwise, add it to the MST. Repeat until there are (V-1) edges in the MST, where V is

the number of vertices in the graph. Prim's algorithm: Start with an arbitrary vertex, say vertex 0, and add it to the MST. Find the edge with the smallest weight that connects a vertex in the MST to a vertex outside of the MST. Add the vertex on the other end of the edge to the MST. Repeat steps until there are (V-1) edges in the MST, where V is the number of vertices in the graph. In terms of time complexity, Kruskal's algorithm has a time complexity of O(E log E), where E is the number of edges in the graph. This is because the algorithm requires sorting all the edges and performing a Union-Find operation for each edge. On the other hand, Prim's algorithm has a time complexity of O(E log V), where V is the number of vertices in the graph. This is because the algorithm requires maintaining a priority queue of edges and performing a decrease-key operation for each edge. In the dynamic case where edges or vertices may be inserted or deleted continuously, Prim's algorithm is better suited to handle a batch of changes. This is because adding or deleting an edge can be done by simply updating the priority queue and performing a decrease-key operation. On the other hand, Kruskal's algorithm would require a complete re-sorting of all edges and a complete re-Union-Find operation for each edge added or deleted. Therefore, Prim's algorithm would be more efficient for handling changes in a dynamic graph.

8

Explain how binary search trees become 2-3 trees and then how 2-3 trees are generalized top B-trees. When does the B-tree grow one level? Explain how you split a node. Write algorithms to insert a key, search a key, and their $O()$. Explain how to exploit the B-tree to compute the set operations above.

8.1 Solution

Binary search trees are data structures made up of a finite set of nodes that is either empty or consists of a node called a root joining two other binary trees, also called sub-trees. In a BST, all elements stored in the left subtree of a node of value K have values $<$ K, and all elements stored in the right subtree of this same node K have values $\geq$ K.

In the other hand, 2-3 trees are a generalization of BST trees where they have the same basic definition of binary trees, but in addition each node contains one or two keys, every internal node has either two or three children, and all leaves are at the same level in the tree i.e. the tree is always balanced. To convert a BST into a 2-3 tree one needs to modify the nodes of the tree to have 2 or 3 keys and 2 or three children. If each node has two keys and two children, simply add a new node (including to the root node) and make the original node a child of the new node. The advantage of the extension of the BST is that it can update easily at a very lower cost.

Moreover, an even better extension is the generalization of 2-3 trees called B-trees. A B-tree has a root that is either a leaf or has at least two children. Each node, except for the root and the leaves, has between $m/2$ and m children. All leaves are at the same level and the tree is always balanced. To covert a 2-3 tree into a B-tree, the original tree can be optimized by allowing each node to have a variable number of keys and children, and not just 2-3. Thus, a B-tree takes as a parameter B the number of children and keys a node can have. The tree will grow by one level when a node is split when inserting a key into a node that already has the maximum number of keys specified by B.

The process of splitting a node follows by taking the middle key and moving it with the parent node, and creating two new nodes with the remaining keys and children. If the parent node has the maximum B keys, then also split the parent node once again repeating the process if necessary.

Short Algorithm to search for keys:
a. Start at the root node
b. Do a binary search on keys in current node.
c. If key is found, return it.
d. Otherwise, follow the branch and recursively repeat the process.

Short Algorithm to Insert keys:
a. Find the leaf node where the key to be inserted belongs to
b. If the leaf node does not exceed the max B keys, insert the node
c. Else if the leaf node exceeds the max B keys, then split the node and move the middle key to the parent node
d. If the parent node is full repeat the process and split the root node moving the middle key to its parent

The asymptotic cost of search, insertion and deletion of nodes in a B-tree is $\theta(log(n))$.

B-trees are useful to compute a variety of applications including the set operations mentioned above such as union, or intersection of two sets. The way it would work for a union operation between n sets would be storing the sets in different B-trees and running them in parallel while comparing each key from each tree. For this example, if n = 2 every two keys would be added to a third set which holds the union of the sets. Similarly, for the intersection of two sets, every two keys can be compared and if they match, it is added to a third set holding the intersection of both sets.

9

For a graph G determine which vertices are reachable from any vertex. The output should be a binary matrix (1=reachable). Show best, average, and worst

cases for time complexity, considering graphs of different densities.

9.1 Solution

To determine which vertices are reachable from any vertex is to determine if there is a path between any two vertices in a graph. This is an All-Pairs Shortest Path problem, which can be implemented by running a single-source shortest path algorithm n times. The time complexity for using Dijkstra's algorithm for this problem is $\theta(n * |E| \, logn)$, however; if the graph is dense it proves to be inefficient, and it will only work if there are no negative weights.

In fact, one of the most efficient algorithms to solve this problem is the Floyd-Warshall algorithm, which has time complexity $\theta(n^3)$. Even though it seems to have a lower time complexity than Dijkstra's, it is the most efficient because it will also work with negative weights and its time complexity will not change considerably if the graph is sparse or dense. It is a dynamic programming-based algorithm for computing the shortest distance, but in this example was modified to determine which vertices are reachable from any vertex.

9.2 Algorithm

Algorithm 4 Modified Floyd-Warshall
Input: An adjancey matrix A of a directed, weighted graph G
Output: A binary matrix B, 1 = reachable, 0 = not reachable

```
1: function IsReachable(A)
2:     D = G                                  ▷ Set D to the adjacency matrix of G
3:     for k = 1 to n do
4:         for  i = 1 to n do
5:             for j = 1 to n do
6:                 D[i][j] = {D[i][j] D[i][k] + D[k][j] }
7:
```

9.3 Time Complexity Analysis

In the average case scenario, the time complexity is $O(n^3)$, where n is the number of vertices V in G. That follows from the formulation of sub-problems for this algorithm where for every pair of vertices i and j, there are n sub-problems, $1 \leq k \leq n$, and to solve each sub-problem requires constant time. Indeed, in the average case scenario this algorithm may run faster or slower depending on how the edges are arranged in the graph. For example, if the edges are

In the best case scenario, when the graph G is sparse, the time complexity is also $O(n^3)$. However, it can run faster than that because there are less pairs of vertices to examine. It will also just depend on how the edges are arranged in

the graph resulting in faster running times or $O(n^3)$.

In the worst case scenario, when G is dense, the time complexity for this algorithm is again $O(n^3)$. That is the case because the algorithm still has to analyze every pair of vertices in G. Denser graphs can have better running times than that, but that will depend on the arrangement of edges as explained in the previous scenarios.

10

For an undirected graph G write an algorithm that enumerates all triangles in the graph, without repetition. Justify all triangles are found (no triangle missing). Show best, average, and worst cases for time complexity, considering graphs of different densities.

10.1 Solution

In order to determine the triangles in a graph one needs to determine the neighboring vertices (u,w) of a given vertex v. Moreover, if there is an edge between u and w, then (u,v,w) forms a triangle.

10.2 Algorithm

Algorithm 5 Enumerate Triangles
Input: An undirected graph G with n vertices and m edges
Output: An enumeration of triangles as a list

```
1: function TRIANGLEENUM(G)
2:     T = []      ▷ T is initially an empty list of vertices representing triangles
3:     for each v in G do
4:         for each neighboring vertex u of v do
5:             for each neighboring vertex w of v  do
6:                 if w is a neighbor of u then
7:                     T = T ∪ (u, v, w)
8:                 end if
9: return T
10:
```

10.3 Analysis and Time Complexity

The reason why all triangles are found and no repetition occurs is because of the order the vertices are output. That is, in the above algorithm function every triangle is stored as (u, v, w), where vertex v is between u and w, which means when evaluating v, the vertices u and w will be only output for v because they

are the neighboring vertices of v.

In the worst case scenario, when the graph is dense the algorithm will traverse over every node of the graph to try to determine if there is a triangle among vertices (u, v, w) or not. Because there are n^3 possible evaluations for every iteration and because of the three nested loops of the function, the worst case running time is O(n^3).

The best case is when the graph is sparse and there are few vertices to evaluate, or if the graph has no triangles. In this case the time complexity is O(n$*$m) since there are n vertices and m edges.

Moreover, the average case for this algorithm could also be a factor of O(n^3), but that is way more complex to evaluate because there are many factors that could change its time complexity, including the arrangements of the edges and the amount of triangles in the graph.