Assignment 4

Kleinberg and Tardos Greedy Algorithms

Ehu Shubham Shaw
Joe Johnson
March 13, 2025

1. (10 points) Exercise 4.1

Question: Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary connected, undirected graph with a distinct cost, c(e) on every edge, e. Suppose e is the cheapest edge in G; that is, c(e*) < c(e) for every edge, e !=4.261 e*. Then, there is a minimum spanning tree T of G that contains the edge, e*.

Answer: True.

Explanation: This statement is correct, and it can be demonstrated using the features of Kruskal's approach for determining the minimum spanning tree. Kruskal's algorithm arranges all edges in ascending order based on their weights, then repeatedly adds the cheapest available edge to the MST as long as it does not form a cycle. Kruskal's algorithm will prioritize e* since it is the cheapest edge in graph G. When e* is assessed, it cannot form a cycle, hence it will undoubtedly be included in the MST. Alternatively, we can consider the cut property: For any partition of vertices into two non-empty sets, the minimum weight edge crossing the partition must be in any MST. If we partition the vertices of e* such that u is in one set and v is in another, then e* is the minimum weight edge crossing this partition, so e* must be included in any MST.

2. (15 points) Exercise 4.2

Question: For each of the following two statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

(a) (8 points)

Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph, G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost ce by its square, ce^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance.

Answer: True.

Explanation: This assertion is correct. When all edge costs are positive and distinct, squaring them preserves the edges' relative order. Specifically, if ce base1 < ce base2 prior to squaring, then ce base1 ^1 < ce base1 ^ 2 post-squaring. Minimum spanning tree techniques, such as Kruskal's or Prim's, rely on the relative ordering of edge weights rather than their absolute values. Because squaring preserves this ordering (where all costs are positive), the algorithms will produce the same tree T, which will remain the minimal spanning tree for the next instance.

(b) (7 points)

Suppose we are given an instance of the Shortest s-t Path Problem on a directed graph, G. We assume that all edge costs are positive and distinct. Let P be a minimum-cost s-t path for this instance. Now suppose

we replace each edge cost ce by its square, ce^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? P must still be a minimum-cost s-t path for this new instance.

Answer: False.

Counterexample: Consider a directed graph with three nodes s, a, and t, and the following edges:

- Edge (s, a) with cost 1
- Edge (a, t) with cost 10
- Edge (s, t) with cost 15

The shortest path from s to t is P_1 = s —> a —> t with total cost 1 + 10 = 11, rather than the direct path P base 2 = s —> t with cost 15.

After squaring the costs:

- Edge (s, a) has cost 1^2 = 1
- Edge (a, t) has cost 10^2 = 100
- Edge (s, t) has cost 15^2 = 225

Now the shortest path from s to t is P base 2 = s —> t with total cost 225, rather than P base 1 = s —> a —> t with total cost 1 + 100 = 101.

This counterexample shows that squaring edge costs can change which path is the shortest, so the statement is false.

3. (15 points) Exercise 4.3

Question: You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight, wi. The trucking station is quite small so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of the greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

Answer:

I'll prove that the current greedy algorithm (packing boxes in arrival order, sending trucks when the next box doesn't fit) minimizes the number of trucks needed.

Proof by : "staying ahead" argument**:**

Let's consider a sequence of boxes with weights w1, w2, ..., wn that arrive in that order. We have a weight limit W for each truck. let's denote the greedy solution as G, which packs boxes in the order they arrive and sends a truck whenever the next box doesn't fit. let OPT be any optimal solution that minimizes the number of trucks while respecting the constraint that boxes must be shipped in the order they arrive.
We will show that G "stays ahead" of OPT in the following sense: After packing the first k boxes, the number of trucks used by G is less than or equal to the number of trucks used by OPT.

Base case: For k = 0 (no boxes), both G and OPT use 0 trucks.

Inductive step: Assume that after packing the first j boxes, G has used at most as many trucks as OPT. Let's consider what happens when we pack the (j+1)th box.

There are two cases:

Case 1: The (j+1)th box fits in the current truck in G's solution. In this case, G adds this box to the current truck and doesn't need a new truck. So G still stays ahead of or equal to OPT.

Case 2: The (j+1)th box doesn't fit in the current truck in G's solution. This means the current truck in G's solution has a total weight greater than W - w_{j+1}. Since G has packed boxes greedily up to this point, the truck is as full as possible with the first j boxes now, OPT also needs to pack the same first j boxes in the same order. If OPT has used more trucks than G for the first j boxes, then G is already ahead. If OPT has used the same number of trucks as G for the first j boxes, then OPT's current truck cannot contain more boxes than G's current truck because G packed as many as possible in order. since the (j+1)th box doesn't fit in G's current truck, it also won't fit in OPT's current truck. So OPT also needs to start a new truck for the (j+1)th box. After packing the (j+1)th box, both G and OPT will have used the same number of trucks.
By induction, after packing all n boxes, G uses at most as many trucks as OPT. Since OPT is optimal, G must use exactly the same number of trucks as OPT, which means G is also optimal.
Therefore, the greedy algorithm currently in use minimizes the number of trucks needed.

4. (15 points) Exercise 4.5

Question: Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Answer:

This problem can be solved using a greedy approach. Here's an efficient algorithm:

- Sort all houses by their positions along the road (from west to east).
- Place the first base station 4 miles east of the westernmost house.
- Iteratively, for each house, check if it's covered by the most recently placed base station:
  - If it is (distance <= 4 miles), continue to the next house.
  - If it's not covered (distance > 4 miles), place a new base station 4 miles east of this house.
- Return the locations of all placed base stations.

pseudocode:

```
Algorithm PlaceBaseStations(houses):
   Sort houses by position from west to east
   baseStations = []
   i = 0
   while i < houses.length:
      station = houses[i] + 4
       baseStations.append(station)
      while i < houses.length and houses[i] <= station + 4:
          i = i + 1

   return baseStations
```

Proof: my algorithm is correct because of below reasons

1. Each base station is placed to cover the leftmost uncovered house plus as many subsequent houses as possible.
2. By placing a station 4 miles east of an uncovered house, we ensure this house is covered within 4 miles.
3. This is optimal because any house more than 8 miles east of the previous base station would require a new station.

Time Complexity & Space Complexity:

- Sorting the houses takes O(n log n) time, where n is the number of houses.
- The greedy placement loop takes O(n) time.
- Overall time complexity: O(n log n).
- O(n) in the worst case for storing the positions of base stations.

5. (15 points) Exercise 4.8

Question: Suppose you are given a connected graph, G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Answer:

I'll prove that a connected graph G with distinct edge costs has a unique minimum spanning tree (MST).

Proof by contradiction:

Assume that G has two different minimum spanning trees, T1 and T2. Since T1 and T2 are different, there exists at least one edge e that is in T1 but not in T2.

When we add e to T2, we create a cycle C in T2 union {e}. Since T2 is a spanning tree, for every edge in this cycle that's not e, it must belong to T2.

Now, in this cycle C, there must be at least one edge e' that is not in T1.

Let's consider what happens if we:

- Remove e from T1
- Add e' to T1 - {e}

This would create a new spanning tree T1', since:

- We're removing one edge from a cycle in T1 union {e'}, so we maintain connectivity
- We're keeping the number of edges the same (|V| - 1), so it remains a tree

If c(e) < c(e'), then T1' would have a lower total cost than T1, contradicting the fact that T1 is an MST. If c(e) > c(e'), then T2 would have a lower total cost than T1, again contradicting that T1 is an MST.
Since all edge costs are distinct, we must have either c(e) < c(e') or c(e) > c(e'). We've shown both cases lead to contradictions. Therefore, our original assumption that there are two different MSTs must be false.
Thus, G has a unique minimum spanning tree.

6. (15 points) Exercise 4.9

Question: One of the motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum total cost. Here we explore another type of objective: designing a spanning network for which the most expensive edge is as cheap as possible.

Specifically, let G = (V, E) be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let T = (V, E') be a spanning tree of G; we define the bottleneck edge of T to be the edge of T with greatest cost.

A spanning tree T of G is a minimum-bottleneck spanning tree if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) (7 points)

Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.

Answer 6(a) : false.

Counterexample: Consider a graph G with 4 vertices (A, B, C, D) and the following edges:

- Edge (A, B) with cost 1
- Edge (B, C) with cost 2
- Edge (C, D) with cost 3
- Edge (D, A) with cost 4
- Edge (B, D) with cost 5

The minimum spanning tree includes edges (A, B), (B, C), and (C, D) with a total cost of 1 + 2 + 3 = 6. The bottleneck edge in this MST is (C, D) with cost 3.

Now consider another spanning tree T' consisting of edges (A, B), (B, C), and (D, A) with a total cost of 1 + 2 + 4 = 7. The bottleneck edge in T' is (D, A) with cost 4.

T' is not a minimum-bottleneck spanning tree because its bottleneck edge cost (4) is higher than the MST's bottleneck edge cost (3).

This shows that not every minimum-bottleneck tree is an MST.

(b) (7 points)

Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.

Answer for 6(b): True.

Proof: I'll prove this by contradiction.

Assume that there exists a minimum spanning tree T that is not a minimum-bottleneck spanning tree. Then there must exist another spanning tree T' with a cheaper bottleneck edge.

Let e be the bottleneck edge of T, and let e' be the bottleneck edge of T'. By our assumption, c(e') < c(e). When we add e' to T (if it's not already in T), we create a cycle. In this cycle, there must be an edge f that is not in T' (otherwise, T' would contain a cycle). If we remove f from T and add e', we get a new spanning tree T*.

Now there are two cases to consider:

1. If f = e (the bottleneck edge of T): Then T* has a lower total cost than T (since c(e') < c(e)), which contradicts T being an MST.
2. If f != e: Then f must have a cost less than e (since e is the bottleneck edge of T). But then T* would have a higher total cost than T (since we're removing a cheaper edge f and adding a more expensive edge e'), which means T' would have an even higher total cost. But this contradicts the assumption that T' has a cheaper bottleneck edge.

Both cases lead to contradictions, so our initial assumption must be false. Therefore, every MST of G is also a minimum-bottleneck spanning tree.

7. (15 points) Exercise 4.21

Question: Let us say that a graph G = (V, E) is a near-tree if it is connected and has at most n + 8 edges, where n = |V|. Give an algorithm with running time O(n) that takes a near-tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all the edge costs are distinct.

Answer:
To solve this problem, I'll leverage the fact that a near-tree has very few extra edges compared to a spanning tree.

1. Initialize an empty set T that will eventually contain the edges of our MST.
2. Identify all cycles in the graph G. Since G has at most n + 8 edges and a tree has n - 1 edges, there are at most (n + 8) - (n - 1) = 9 extra edges, which means at most 9 cycles.
3. For each cycle, find the maximum cost edge and mark it for deletion (we'll never include it in the MST).
4. Include all unmarked edges in T.
5. Return T as the MST.

Pseudocode:

```
Algorithm FindMST(G = (V, E)):
  T = {}
  visited = set()
   parent = {}
   cycles = []

   for each vertex v in V:
      if v not in visited:
         DFSFindCycles(v, visited, parent, -1, cycles, E)
   edgesToDelete = set()

   for each cycle in cycles:
      maxCostEdge = edge with maximum cost in cycle
      edgesToDelete.add(maxCostEdge)

  for each edge e in E:
      if e not in edgesToDelete:
         T.add(e)

   return T

Function DFSFindCycles(v, visited, parent, prev, cycles, E):
   visited.add(v)
   parent[v] = prev

   for each neighbor u of v in G:
      if (v, u) in E:
         if u not in visited:
            DFSFindCycles(u, visited, parent, v, cycles, E)
         else if u != parent[v]:
           cycle = constructCycleFromBackEdge(v, u, parent)
            cycles.append(cycle)
```

Space Complexity & Time Complexity Analysis:

- DFS on a graph with n vertices and m edges takes O(n + m) time.
- In a near-tree, m <= n + 8, so the DFS takes O(n) time.
- Finding and processing all cycles takes O(n) time since there are at most 9 cycles.
- Overall time complexity: O(n).
- O(n) for storing the visited set, parent mapping, and the final MST.

So this algorithm correctly finds an MST because:

1. It follows Kruskal's algorithm intuition: we avoid the maximum cost edge in each cycle.
2. For each cycle, the maximum cost edge will never be part of an MST.
3. With distinct edge costs, there is a unique MST.

The algorithm efficiently exploits the near tree property to achieve linear time complexity, as required.