HW 3 – Greedy Algorithms, Local Search

CS 6150

Submission date: Friday, Oct 6, 2023, 11:59 PM

This assignment has 4 questions for a total of 50 points, plus 4 bonus points (earning bonus points will help your grade for points lost here or in other assignments). Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Set Cover Revisited	16	
Selling Intervals	12	
Forming pairs	10	
Finding diverse elements	12	
Total:	50	

Question 1: Set Cover Revisited ..... **[16]**

In class, we saw the set cover problem (phrased as picking the smallest set of people who cover a given set of skills). Formally, we have n people, and each person has a subset of m skills. Let the set of skills of the ith person be denoted by S_i, which is a subset of $[m]$ (shorthand for $\{1, 2, \ldots, m\}$).

(a) **[6]** As before, suppose that there is a set of k people whose skill sets cover all of $[m]$ (i.e., together, they possess all the skills). Now, suppose we run the greedy algorithm discussed in class for $4k$ iterations. Prove that the set of people chosen cover $> 90\%$ of the skills.

Answer:

To prove that the set of people chosen cover $> 90\%$ of the skills, we can simply prove that the number of skills not covered by people chosen is $< 10\%$ of the skills.
From the discussion in class, we know that $u_t \leq u_0(1 - \frac{1}{k})^t$, where u_t is the number of skills yet to be covered after step t, and u_0 is the number of skills that we start with.

Let $t = 4k$. Substituting, we have $u_{4k} \leq u_0(1 - \frac{1}{k})^{4k}$
Bernoulli's inequality states that $(1 + x)^n > 1 + nx$
$\implies (1 + \frac{1}{k-1})^k \geq 1 + \frac{k}{k-1}$
$\implies 1 + \frac{k}{k-1}$ will always be > 2
$\implies$ Reciprocal of $1 - \frac{1}{k}$ is $1 + \frac{1}{k-1}$
$\implies (1 - \frac{1}{k})^k < \frac{1}{2}$
$\implies u_{4k} \leq u_0(1 - \frac{1}{k})^{4k}$ simplifies to $u_{4k} < 2^{-4}.u_0 < 10\%u_0$
Therefore we have the number of skills not covered by people chosen is $< 10\%$ of the skills and thus the set of people chosen cover $> 90\%$ of the skills.

(b) **[4]** Consider the following "street surveillance" problem. We have a graph (V, E) with n nodes and m edges. We are allowed to place surveillance cameras at the nodes. Once placed, they can monitor all the edges incident to the node. The goal is to place as few cameras as possible, so as to monitor **all the edges** in the graph. Show how to cast the street surveillance problem as Set Cover.

Answer:

We can easily cast this problem as a set cover problem by taking the n nodes to represent "people" and m edges to represent "skills". We say a vertex v covers an edge e if v is incident on e. Formally, $V = \{v_1, v_2, ..., v_n\}$ be the set of vertices. Then we define $S_{v_i} = \{e \in E | e = v_i y (y \in V)\}$. The set cover problem is to find $U \subseteq V$ such that $\bigcup_{u \in U} S_u = E$ and $|U|$ is minimized.

(c) **[6]** Let (V, E) be a graph as above, and suppose that the optimal solution for the street surveillance problem places k cameras (and is able to monitor all edges). Now consider the following "lazy" algorithm:

1. initialize $S = \emptyset$
2. while there is an unmonitored edge $\{i, j\}$:
   add both i, j to S and mark all their edges as monitored

Clearly (due to the while loop), the algorithm returns a set S that monitors all the edges. Prove that the set also satisfies $|S| \leq 2k$ (recall that k is the number of nodes in the optimal solution).

MORAL. Even though the algorithm looks "dumber" than the greedy algorithm, it has a better approximation guarantee — 2 versus $\log n$.

Hint. Consider the edges $\{i, j\}$ encountered when we run the algorithm. Could it be that the optimal set chooses *neither* of $\{i, j\}$?

Answer:

We can see that since the lazy algorithm picks both endpoints of an edge it is essentially picking an edge. Let F be the set of edges picked by the lazy algorithm. We can see that for any edges $e, f \in F$, the endpoints of e and f are distinct. Without loss of generality we may assume the algorithm picks vertex e before vertex f, and since we mark all the edges incident to endpoints of e as monitored, so the unmonitored edges picked after that could not be incident to endpoints of e. Thus f is vertex disjoint from e. We now we have a set of vertex disjoint edges.

Since the vertex cover has to cover these edges, then the vertex cover should have at least one endpoint from each edge in F, and thus $|F| \leq k$. We know that $|S| = 2|F|$ since for each edge we have two distinct vertices as we have argued above. Therefore, $|S| = 2|F| \leq 2k$ as desired.

Question 2: Selling Intervals ..... **[12]**

Suppose we are given n intervals on the real line — $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$. Of course, some of the intervals overlap. For each interval, we have a buyer, who is willing to pay a price of p_i for interval i. The goal is to find the best subset of intervals to sell, such that (a) intervals sold are all non-overlapping, and (b) the total price is maximized.

(a) [**4**] It is natural to want to sell intervals that are short but have a high price. Consider a greedy algorithm that first sorts the intervals in decreasing order based of the ratio of price to length, i.e., $p_i/(b_i - a_i)$, and then does the following:

1. initialize $S = \emptyset$
2. go over the intervals in sorted order (as above), doing the foll:
   add the interval to S if it does not overlap with any interval already in S

Give an instance in which the greedy choice is sub-optimal.

Answer:
The greedy algorithm above does not take into account the lengths. Suppose we have two intervals: $[1, 3]$ and $[2, 10]$. Suppose the first has price 4 and the second has price 8. The price-to-length ratio is better for the first, but the optimal choice is to pick the second.

(b) [**8**] Design an algorithm based on dynamic programming. Your algorithm should run in time polynomial in n. As always, include an analysis of correctness and the running time.

Answer:
Let us start by sorting the n intervals by increasing order of end point of the interval. At every interval j, the dynamic programming solution considers 2 options, either to pick the j^{th} interval, or to not pick it. It then computes the price paid by the buyer at each choice and picks the best of the two choices.
Consider the optimal solution (OPT) for a subproblem: $OPT(j)$ = the optimal solution considering only intervals $1, ..., j$
Let $f(j)$ be the largest value of $i < j$ such that the i^{th} interval does not overlap with the j^{th}. Given a j^{th} interval, we perform binary search on intervals 1 to $j - 1$ to find the rightmost interval that does not overlap with j. The property of the intervals being sorted by their endpoints ensures that if an interval at a position i overlaps with j, then all intervals after i will also overlap.

Base Case: if $n == 0$ return 0
Dynamic programming dependency relation:

$$OPT(j) = max \begin{cases} p_j + OPT(f(j)), & if \ \ j \in OPT \\ OPT(j-1), & if \ \ j \notin OPT \\ 0 \quad else \end{cases}$$

Time Complexity: The intervals are initially sorted by their endpoints. This takes $O(n \log n)$. For each interval j, we can calculate $f(j)$ in $log(n)$ time using binary search since intervals are sorted by their endpoints. Computing $OPT(j)$ for all j is $O(n)$. This algorithm runs in overall time $O(nlogn)$.

Correctness: The algorithm exhibits the optimal substructure property because it builds the solution for each interval j based on the optimal solutions of smaller subproblems. By finding the latest non-overlapping interval that ends before j and considering the maximum value among those options, it ensures that the optimal solution for interval j is based on the optimal solutions for smaller subintervals. By considering both including and excluding the current interval and choosing the option with the maximum value, the algorithm ensures that it selects the best possible set of non-overlapping intervals that maximize the total value.

Pseudocode:

(c) [] **[4] Bonus points**: You are given the same set of intervals with associated prices, and you still want to maximize the total price. However, you are allowed to sell overlapping intervals, but for

Algorithm 1 Weighted Interval Scheduling

```
function LatestNonOverlapping(intervals, j):
    for i ← j − 1
downto 0 do
        if intervals[i][1] ≤ intervals[j][0] then
            return i
    return −1
function WeightedIntervalScheduling(intervals, n):
    Sort intervals in increasing order of finish time
    Initialize an array OPT of size n + 1
    OPT[0] ← 0
    for j ← 1 to n do
        val ← interval[j − 1]
        non_overlapping ← latestNonOverlapping(interval, j − 1)
        if non_overlapping ≠ −1 then
            val ← val + OPT[non_overlapping + 1]
        OPT[j] ← max(val, OPT[j − 1])
    return OPT[intervals]
```

each overlapping interval, you incur a penalty cost of "penalty" dollars. Your goal is to find the subset of intervals to maximize the total price while minimizing the penalty cost.

Provide a dynamic programming solution for this extended problem. Analyze the time complexity of your solution.

Sort the intervals by their end time. The dynamic programming solution for this problem requires a 2D memoization table. The first dimension represents the position of a subinterval on the real line. Given n original intervals, there can be up to $2n - 1$ subintervals, created by considering the start and end of each original interval as defining points. The second dimension comes from the set of of overlapping intervals. For each subinterval, there's a set of original intervals, $J(i)$, that overlaps with it. Since each interval in this set may or may not overlap with the subinterval, this leads to a binary decision for each member of the set. Thus, the number of potential configurations for overlaps in $J(i)$ is $2^{|J(i)|}$, leading to the exponential size of this dimension in the worst-case scenario. The overlap configuration in this case refers to a specific combination of original intervals that overlap with a given subinterval.

The time complexity is $O(n \times 2^n)$, as that is the dimension of the DP table.

Question 3: Forming pairs ..... **[10]**

Consider n people, each having a *skill level* ($s_1, s_2, \ldots, s_n$, which are integers). We are also given a threshold T. Two people form a *successful pair* if the sum of their skill levels is $\geq T$.

The goal is to form the maximum number of successful pairs. For convenience, suppose that s_i are already sorted in increasing order. Now, consider the following greedy algorithm: given a set of people, consider pairing the least skilled person and the most skilled person. If the sum of the skill levels is $\geq T$, form the pair and recurse; else, delete the least skilled person and recurse on the rest.

(a) **[3]** Show how to implement this algorithm in $O(n)$ time (by writing pseudo-code and performing a runtime analysis).

Answer:

Algorithm 2 Find Successful Pairs

```
1:  function FindSuccessfulPairs(S, T):
2:      begin ← 1
3:      end ← n
4:      output ← ∅
5:      while begin < end do
6:          if S[begin] + S[end] ≥ T then
7:              begin ← begin + 1
8:              end ← end − 1
9:              output.append({begin, end})
10:         else
11:             begin ← begin + 1
12:     return output
```

Time complexity is $O(n)$ because in each iteration we look at at least one element, and we look at each element at least once. So the total running time is $O(n)$.

(b) **[7]** Does the algorithm always output the pairing with the largest possible number of (successful) pairs? Either provide a counter-example, or give a formal proof of correctness.

Answer:

To prove the correctness of the given greedy algorithm, we can use a proof by induction on the number of people in the set.

Base Case ($n = 2$): When there are only two people, the algorithm simply checks if their combined skill level is greater than or equal to the threshold T. This is an optimal choice because there's only one possible pair to form, and it maximizes the number of successful pairs if the sum is greater than or equal to T.

Inductive Hypothesis: Assume that the algorithm produces an optimal solution for a set of k people, where $k > 2$.

Inductive Step: We want to show that the algorithm produces an optimal solution for a set of $k+1$ people.

Let's denote the skill levels of the k+1 people as $s_1, s_2, ..., s_k, s_{k+1}$, where $s_1 \leq s_2 \leq ... \leq s_k \leq s_{k+1}$.

When considering the $k+1$ people, the algorithm makes the following choices:

If $s_1 + s_{k+1} \geq T$ (i.e., the least skilled person and the most skilled person can form a successful pair), the algorithm pairs them and then solves the problem for the remaining k people. If $s_1 + s_{k+1} < T$, the algorithm removes s_1 and solves the problem for the remaining k people. We need to prove that both of these choices result in an optimal solution for the set of $k+1$ people.

Case 1: Pairing s_1 and s_{k+1}

If we pair s_1 and s_{k+1}, we create a successful pair. This does not affect the optimal solution for the remaining $k-1$ people because pairing s_1 with s_{k+1} does not prevent us from forming any other possible successful pairs among the remaining $k-1$ people. Therefore, this choice is valid.

Case 2: Removing s_1

If we remove s_1, it is possible that we lose the opportunity to pair s_1 with some other person in the group. However, since we are removing the least skilled person, it is guaranteed that $s_1 + s_i < T$ for all i in $[2, k]$, and hence, there are no successful pairs involving s_1 and the other$k-1$ people. Removing s_1 does not prevent us from forming any other possible successful pairs among the remaining k people. Therefore, this choice is valid.

Since both choices (pairing s_1 and s_{k+1} or removing s_1) are valid, and we assume that the algorithm produces an optimal solution for k people, the algorithm will also produce an optimal solution for $k+1$ people.

Question 4: Finding diverse elements ..... **[12]**

A common problem in returning search results is to display results that are *diverse*. A simplified formulation of the problem is as follows. We have n points in Euclidean space of d-dimensions, and suppose that by distance, we mean the standard Euclidean distance. The goal is to pick a subset of k (out of the n) points, so as to maximize the sum of the pairwise distances between the chosen points. I.e., if the points are denoted $P = \{p_1, p_2, \ldots, p_n\}$, then we wish to choose an $S \subseteq P$, such that $|S| = k$, and $\sum_{p_i, p_j \in S} d(p_i, p_j)$ is maximized.

A common heuristic for this problem is local search. Start with some subset of the points, call them $S = \{q_1, q_2, \ldots, q_k\} \subseteq P$. At each step, we check if replacing one of the q_i with a point in $P \setminus S$ improves the objective value. If so, we perform the swap, and continue doing so as long as the objective improves. The procedure stops when no improvement (of this form) is possible. Suppose the algorithm ends with $S = \{q_1, \ldots, q_k\}$. We wish to compare the objective value of this solution with the optimum one. Let $\{x_1, x_2, \ldots, x_k\}$ be the optimum subset.

(a) **[3]** Use local optimality to argue that:

$$d(x_1, q_2) + d(x_1, q_3) + \cdots + d(x_1, q_k) \leq d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k).$$

Answer: Local optimality implies that you cannot improve the objective value by swapping one of the points in the current solution set $S = q_1, q_2, \ldots, q_k$ with a point from $P \setminus S$. Let's focus on $q_1 \in S$ and $x_1 \in P \setminus S$. Swapping q_1 with x_1 would mean replacing q_1 with a point that is already in the optimal solution $x_1, x_2, \ldots, x_k$. Since the objective value cannot be improved, it means that:

$$d(x_1, q_2) + d(x_1, q_3) + \cdots + d(x_1, q_k) \leq d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)$$

which proves the argument.

(b) **[4]** Deduce that: [*Hint:* Use two inequalities of the form above.]

$$(k-1) \cdot d(x_1, x_2) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

From the first part,

$$d(x_1, q_2) + \cdots + d(x_1, q_k) \leq d(q_1, q_2) + \cdots + d(q_1, q_k)$$

Similarly, considering x_2, we get:

$$d(x_2, q_2) + \cdots + d(x_2, q_k) \leq d(q_1, q_2) + \cdots + d(q_2, q_k)$$

Adding the two inequalities we have:

$$d(x_1, q_2) + d(x_2, q_2) + \cdots + d(x_1, q_k) + d(x_2, q_k) \leq 2\left[d(q_1, q_2) + \cdots + d(q_1, q_k)\right]$$

Due to the triangular inequality, we have that:

$$d(x_1, x_2) \leq d(x_1, q_i) + d(x_2, q_i)$$

If we apply the triangular inequality to each two terms on the left involving the same q_i we have:

$$(k-1)d(x_1, x_2) \leq d(x_1, q_2) + d(x_2, q_2) + \cdots + d(x_1, q_k) + d(x_2, q_k) \leq 2\left[d(q_1, q_2) + \cdots + d(q_1, q_k)\right]$$

Which proves that:

$$(k-1)d(x_1, x_2) \leq 2\left[d(q_1, q_2) + \cdots + d(q_1, q_k)\right]$$

(c) [**5**] Use this expression to argue that

$$\sum_{i,j} d(x_i, x_j) \leq 2 \sum_{i,j} d(q_i, q_j).$$

Note that this shows that the locally optimum solution has objective value at least $1/2$ the optimum.

We have seen from the previous answer that:

$$(k-1)d(x_1, x_2) \leq 2\left[d(q_1, q_2) + \cdots + d(q_1, q_k)\right].$$

that can be rewritten as:

$$(k-1)d(x_1, x_2) \leq 2 \sum_{j \neq 1} d(q_1, q_j).$$

Note that if on the left hand side we replace x_2 with any x_j (with $j \neq 1$) the right hand side does not change. Therefore we can add $k-1$ such terms for all the possible x_j and have:

$$\sum_{j \neq 1} (k-1)d(x_1, x_j) \leq (k-1)2 \sum_{j \neq 1} d(q_1, q_j).$$

dividing both sides by $k-1$ we have:

$$\sum_{j \neq 1} d(x_1, x_j) \leq 2 \sum_{j \neq 1} d(q_1, q_j).$$

This inequality is valid for any x_i, q_i that repalces x_1, q_1. We can therefore write for any fixed i:

$$\sum_{j \neq i} d(x_i, x_j) \leq 2 \sum_{j \neq i} d(q_i, q_j).$$

Adding these inequalities for all possible values of i we have the required solution:

$$\sum_{j,i} d(x_i, x_j) \leq 2 \sum_{j,i} d(q_i, q_j).$$