Course: Algorithm Analysis
Module: Approximation Algorithms

Gunnar Miller

March 22, 2025

1 Similarity Distance of Strings

For any two strings p and q in a set S, a distance $d(p, q) \geq 0$ is defined.

Given a threshold value $\Delta \geq 0$, two strings p and q are *similar* if $d(p, q) \leq \Delta$. A subset R of S is called a representative set of S if for any string $p \in S$, $d(p, q) \leq \Delta$ for some $q \in R$. The minimum representative set problem is to find a representative set of minimum size. Give a polynomial time $O(\log n)$-approximate algorithm for the problem.

Idea: This problem straightforwardly reduces to the set cover problem, allowing us to simply re-use the approximation algorithm for that problem. In particular, each string "covers" the subset of strings of distance Δ or less from itself, so finding a minimal representative set is exactly the same problem as finding a minimal set cover from among the subsets defined by each string. As we only care about minimizing the size of the representative set, all strings are implicitly weighted at 1.

Implementation: We assume we have S implemented as a set of strings, the value of Δ stored in the variable 'Delta' (of whatever appropriate numerical type) and the function $d(p, q)$ implemented under the name 'dist,' taking in two strings and returning a value of the same type as Delta. When the algorithm terminates the set 'rep_set' will hold our representative set.

```
S_covers = {} #dict so we can easily refer to both the set and what it covers
R = S
rep_set = set()
for string1 in S:
    covered = set()
    for string2 in S:
        if dist(string1, string2) <= Delta:
            covered.add(string2)

    S_covers.update({string1: covered})

while len(R) > 0:
    best = 0
    best_string = ""
    for string in S_covers:
        coverage = len(R.intersect(S_covers[string]))
        if  coverage > best:
            best = coverage
            best_string = string

    rep_set.add(best_string)
    R.remove(best_string)
    S_covers.remove(best_string)
```

Analysis:

As our algorithm is exactly the greedy set-cover algorithm given in the textbook, we don't need to re-prove the bound given by 11.11, merely apply it.

Recall that $d*$ is the size of the largest covering set in our approximate solution C; the maximum possible value is simply n (for a set that covers all of S). As all sets are weighted 1, the optimal weight w^* is simply the size of the optimal covering set. The bound given thus reduces to $|C| \leq w^* H(n)$. Recalling that H is the harmonic series and is bounded above by $\log(n)$ we have $|C| \leq w^* \log(n)$ as desired.

Both of the loops in the algorithm have a second inner loop, with both outer and inner loop iterating through a set of size $O(n)$, while applying constant-time operations internally. Thus our time complexity is $O(n^2)$.

2 Greedy Algorithm for Feasible Sets

Given a set of positive integers $A = \{a_1, .., a_n\}$, a positive integer B and a subset $S \subseteq A$, $t(S) = \sum_{a_i \in S} a_i$ is called the total sum of S. A subset S is called a feasible set if $t(S) \leq B$. The maximum feasible set problem is find a feasible set S with $t(S)$ maximized.

2.1 Arbitrarily Poor Approximation

A simple greedy algorithm works as follows: make one pass through A. For each element, add it to S if and only if it wouldn't cause S to be infeasible

Let $B = 100$ and $A = 2, 99$. Clearly the maximum possible $t(s)$ is 99, but the greedy algorithm will return the set $S = 2$ which does indeed have a $t(s)$ of less than half the optimal value.

2.2 Improvement to a 1/2 Approximation

Idea: We improve the algorithm in the previous part by the thoroughly ingenious method of just running it twice (removing elements from A as we go). Surprisingly, this provably produces the desired result.

Implementation: We implement the given greedy algorithm in Python, with the minor change that we remove items from A as we place them in a subset. Assume we start with A implemented as a set and B as a positive int. Our answer is stored in the variable feasible_set, holding whichever of our subsets has the larger total.

```
def limited_sum(starting_set, limit):
    subsum = 0
    subset = set()
    for elem in starting_set:
        if subsum + elem <= limit:
            subset.add(elem)
            starting_set.remove(elem) #removed from A as it is passed by reference
            subsum += elem

    return subset, subsum

S_1, sum_1 = limited_sum(A, B)
S_2, sum_2 = limited_sum(A, B)

if sum_2 > sum_1:
    feasible_set = S_2
else:
    feasible_set = S_1
```

Analysis: The algorithm is clearly $O(n)$ as it makes two linear-time passes through the set A.

To see that it produces a 1/2-approximation, we first need to consider the special case where $t(S) \leq B$. In this case, the first pass places everything in S_1 and thus it is exactly the optimal set.

For the other case, we now assume $t(S) > B$. Now we must have $t(S_1) + t(S_2) > B$; if we did not then all the elements of S_2 could have been added to S_1 on the first pass without exceeding the limit. But for this to be true we must either have $t(S_1) > \frac{1}{2}B$ or $t(S_2) > \frac{1}{2}B$ or both. Obviously $t^* \leq B$, so our final feasible set S' (being whichever of S_1 or S_2 has the larger sum) must have $t(S') > \frac{1}{2}B \geq \frac{1}{2}t^*$.

3 Approximate Greedy Algorithm for Load Balancing

Consider the following load balancing problem: we have m slow machines and k fast machines. We have a batch of n jobs, each which takes a different time $t_1, t_2, ...t_n$ to complete on a slow machine. Fast machines work at twice the speed, so a job that takes t_i on a slow machine will take $t_i/2$ on a fast machine. We wish to assign jobs so as to minimize the time required to finish the whole batch.

A greedy algorithm for this problem is to assign the jobs to machines in arbitrary order 1 to n, assigning each job i to the machine that currently has the minimum load. Prove the approximation ratio for this algorithm.

Solution:

The approximation ratio is 1/3 as we will show below.

Consider the finish time T^* of the optimal solution. No individual machine can have load greater than T^* so we must have $\sum_i t_i \leq (n + 2m)T^*$.

Now consider the finish time T of the solution produced by our algorithm. Obviously out of all our machines, some must be the last to terminate, that is, we must have $T = T_i$ where T_i is the load on some machine i. Let T_{max} be the load of the largest job on that machine: we must have $T_i - T_{max} \leq T_j$ for all $j \neq i$ (or T_{max} would not have been placed on machine i) and we must have $T_j \leq T^*$ for some j or the times for our solution would sum to more than $\sum_i t_i \leq (n + 2m)T^*$. Taken together, this means we must have:

$$T_i - T_{max} \leq T^* \tag{1}$$

Depending on which sort of machine our approximate solution has it processed on, the job with load T_{max} will take at a minimum take time either T_{max} or $T_{max}/2$ to process in the optimal solution, so at best the optimal solution can be half the duration of T_{max}, or equivalently:

$$T_{max} \leq 2T^* \tag{2}$$

Adding (1) and (2) together yields

$$T_i - T_{max} + T_{max} \leq T^* + 2T^* \tag{3}$$

$$T_i \leq 3T^* \tag{4}$$

$$\frac{1}{3}T \leq T^* \tag{5}$$

As desired.

4 Arbitrary Algorithm for Approximate 3-D Matching

Let X, Y and Z be disjoint sets with $|X| = |Y| = |Z|$. Given a set $T \subseteq X \times Y \times Z$ of ordered triples, M is a 3-dimensional matching if $M \subseteq T$ and no element of $X \cup Y \cup Z$ appears in more than one tripe in M. The Maximum 3-D Matching Problem is to find the largest matching possible within T.

Give an approximation algorithm that finds a 3-D matching of at least 1/3 the maximum possible size in polynomial time and analyze the algorithm.

Idea: This problem can be solved trivially when one notices how extremely loose of a requirement a 1/3-approximate solution is to this problem. Literally any algorithm that keeps selecting valid nodes until none are left will meet the requirement.

In implementation, we can enforce the requirement that elements not match by maintaining a set of valid possible choices R, which will originally be all of T. Whenever we add an element to our matching set, we remove all elements that match it in any coordinate.

Algorithm: Assume T is in some arbitrary order. Likewise assume X, Y and Z are in some arbitrary orders.

```
R=T
M = set()

for ele1 in R:
   M.add(ele1)
   for ele2 in R: #loop removes all coordinate matches including ele1 itself
       for coord in ele1:
           if coord in ele2:
               R.remove(ele2)
```

Analysis: To prove the approximation, we need simply notice that adding an element to our matching set M can prevent at most three other elements from later being added to M. Specifically, consider an element of T $p = (x, y, z)$. The set of other elements that clash with p in some coordinate $C = \{(x', y', z') \in T | (x = x') \vee (y = y') \vee (z = z')\}$ may be arbitrarily large. But excluding p from M lets us add at most three of them: one containing x, one containing y and one containing z. So even if our algorithm makes maximally bad choices, and each element we select excludes the maximum possible number of elements from M, the result can be an M no more than 3x as large as the one returned by our algorithm.

The outer loop iterates through the set R, and on each iteration the inner loop also iterates through R, doing a constant amount of work per element. As $|R| \leq |T|$ through the whole runtime the algorithm will be at worst $O(|T|^2)$ (though the average case ought to be significantly better). If we assume $|X| = |Y| = |Z| = n$ then we have $|T| \leq n^3$ for a worst-case time complexity of $O(n^6)$.

5 Approximate Greedy Algorithm for Maximum Independent Set

Suppose we have a graph G, with a one-to-one function $w : V(G) \to \mathbb{Z}$ assigning integer weights to the vertices of G. The maximum-weight independent set problem is to find an independent set $S \subseteq V(G)$ such that the sum of the weights of elements in S is maximized.

Consider the special case in which G has n^2 nodes, each distinctly labeled by an ordered pair $(i, j) \in \mathbb{N} \times \mathbb{N}$ with $i \leq n, j \leq n$. A pair of nodes (i, j) and (i', j') are connected by an edge if and only if either $i = i'$ and $|j - j'| = 1$ or $j = j'$ and $|i - i'| = 1$.

Give a polynomial time 1/4-approximate algorithm for the maximum-weight independent set algorithm in this special case.

Idea: A simple greedy algorithm that selects the node of greatest weight from among the set of nodes still available.

Implementation: Assume our nodes are given in dictionary G of the form $\{(i, j) : w\}$, with the label of the node being the tuple (i, j) and the weight being w.

```
def neighbors(node): #returns coordinates of a node's neighbors
    left = (node[0]-1,node[1])
    right = (node[0]+1,node[1])
    top = (node[0],node[1]+1)
    bottom = (node[0],node[1]-1)
    return [left,right,top,bottom]

R = G
opt = set()

while len(R) > 0:
    biggest = 0
    best = ""
    for node in R:
        if R[node] > biggest:
            biggest = R[node]
            best = node

    opt.add(best)
    for neighbor in neighbors(node):
        R.pop(neighbor,0)
    R.pop(node)
```

Analysis:

Suppose our algorithm picks k nodes in total. We will prove by induction that the set of nodes excluded from S by a given selection can have total weight no more than 4 times the weight of that selection. From this result proving the approximation will follow shortly:

Base Case: Let s_1 be the first node selected. The set of nodes adjacent to s_1 can contain at most four elements, which we will call $E_1 = \{a_1, b_1, c_1\}$ and d_1. The algorithm selects the node with the greatest weight out of all the nodes in the graph, we must have $w(s_1) \geq w(a_1)$ and likewise for b_1, c_1 and d_1. Thus $4w(s_1) \geq \sum_{x \in E_1} w(x)$.

Inductive Step: Suppose our algorithm has already picked the first $i - 1$ nodes for some $i < k$. Consider the set E_i of all nodes newly excluded from S by the ith nodes selection (that is, the set of nodes adjacent to the ith node, but not adjacent to any of nodes 1 through $i - 1$). Clearly E_i can contain at most four elements, which we will call a_i, b_i, c_i and d_i. Our algorithm picks the node with the highest weight among those not yet excluded, so as above we must have $4w(s_i) \geq \sum_{x \in E_i} w(x)$.

From the above, we know for all $i \leq k$, $w(s_i) \geq \frac{1}{4} \sum_{x \in E_i} w(x)$. But of course our algorithm only terminates when all nodes have been either picked (and are in S) or are unavailable to pick (and are in E_i for some i). Thus we must have $V(G) = S \cup E_1 \cup E_2 \cup ... \cup E_k$. Let S^* be the optimal solution; then for each i we can either have $s_i \in S^*$ or some number of the nodes from E_i in S^* but not both. So then we have $\sum_{x \in S^*} w(x) \leq \sum_{i=1}^{k} \max(w(s_i), \sum_{y \in E_i} w(y)) \leq \sum_{i=1}^{k} 4w(s_i)$ or $\frac{1}{4} \sum_{x \in S^*} w(x) \leq \sum_{y \in S} w(y)$ as desired.

6 Approximate 3-SAT via Hamming Distance

Let Φ be a 3-SAT instance of n boolean variables $x_1, x_2, ..., x_n$. A truth assignment f on those variables can be represented as a bit string with the ith bit set to 1 if f assigns x_i to True and 0 if it assigns x_i to false. For a pair of such truth assignments f and f' let $d(f, f')$ be the Hamming distance between their bitstring representations (that is, the number of bits where they differ). For any non-negative integer d we define the algorithm Explore(Φ, f, d) as follows:

Explore(Φ, f, d):
    if f satisfies Φ then return True
    else if $d = 0$ then return False
    else
        let $C = (l_1 \vee l_2 \vee l_3)$ be a clause of Φ that is not satisfied by f
        let f_i $(i = 1, 2, 3)$ be the truth assignment obtained from f by inverting the assigned value to l_i ;
        if Explore$(\Phi, f_1, d-1))$ = True then return True;
        if Explore$(\Phi, f_2, d-1))$ = True then return True;
        if Explore$(\Phi, f_3, d-1))$ = True then return True;
    return NO

Given a positive integer d and a truth assignment Φ, Explore(Φ, f, d) should return True iff there is a satisfying truth assignment for Φ within Hamming distance d of f.

6.1 Correctness and Efficiency of Explore(Φ, f, d):

Prove the algorithm has the stated property and analyze its run time.

Suppose Explore(Φ, f, d) returns True iff there is a satisfying assignment f' with $d(f, f') \leq d$ by induction.

Consider the base case when $d = 0$. In this case, clearly Explore(Φ, f, d) returns True if and only if f satisfies Φ.

For the inductive step, let us assume that for all $d \leq n$, Explore(Φ, f, d) returns YES if and only $\exists f'$ that satisfies Φ with $d(f, f') \leq n$. Suppose there is a satisfying assignment f^* with $d(f, f^*) \leq n + 1$. If $f^* = f$ then Explore$(\Phi, f, n+1)$ returns YES immediately. If instead $d(f^*, f_i) \leq n$ for $i \in 1, 2, 4$ (as defined relative to f in the definition of Explore) then by assumption, the recursive function call Explore(Φ, f_i, n) will return YES and thus so will Explore$(\Phi, f, n+1)$. Conversely, suppose no such f^* exists. Clearly Explore$(\Phi, f, n+1)$ will not immediately return True in this case, and by assumption none of the function calls Explore(Φ, f_i, n) will return True, leaving Explore$(\Phi, f, n + 1)$ to its default behavior of returning False.

The running time of Explore(Φ, f, d) depends in detail on the number of clauses m contained in Φ. With no separate bound on m we can only bound it very weakly in terms of n. In particular, assuming Φ contains only distinct clauses and that the same boolean variable never appears twice in a given clause, there are already $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$ possible ways to pick the variables that appear in a clause. If with further assume that the same trio of variables never appears in two distinct clauses, these leaves us with m being $O(n^3)$ (otherwise m would be exponential in n^3 as there are 2^3 logically distinct clauses that can be formed with each trio).

With this bound, be can analyze the runtime of Explore(Φ, f, d). Checking to see if a truth assignment satisfies Φ takes $O(n^3)$ time (we must examine every clause) but finding an unsatisfied clause may be done concurrently (we simply record some clause that fails). Obtaining a revised truth assignment by modifying this clause takes constant time: we only need to change one element of f. Finally, each call of Explore() may cause three other recursive calls, with a maximum recursion depth of d. This the total run time is $O(n^3 3^d)$.

6.2 Improved Exponential Bound for General 3-SAT

Using the algorithm as a subroutine, design an algorithm that can solve the decision version of the general 3-SAT in $O(p(n)(\sqrt{3})^n)$ run time where $p(n)$ is a polynomial in n.

Let f be a truth assignment on n variables and let $\bar{f}$ be the assignment that results from reversing every boolean variable assignment in f. Consider some other truth assignment f', which differs from f by k variables (and thus from $\bar{f}$ by $n - k$ variables). We either have $k \leq n/2$ in which case $d(f, f') \leq n/2$ or we have $k > n/2$ in which case $d(\bar{f}, f') \leq n/2$.

Thus if we pick any arbitrary f the algorithm:

1. Explore$(\Phi, f, n/2)$
2. Explore$(\Phi, \bar{f}, n/2)$

Is guaranteed to explore every possible clause. Our only setup time comes from constructing $\bar{f}$ from f, which in $O(n)$ time (and thus won't affect our final bound). Referring to the previous part the run time of this algorithm is thus $O(n^3 3^{n/2}) = O(n^3 (3^{1/2})^n) = O(n^3 (\sqrt{3})^n)$ as desired.