

CSE 101 Homework 3
Due Thursday, March 3
Back-Tracking and Greedy Algorithms

Directions: For each of the first four problems, a "high level" greedy strategy is given. For some of the problems, the strategies give a correct (optimal) solution, and for others, it sometimes gives incorrect (suboptimal) solutions. For each, decide whether the greedy strategy guarantees optimal solutions. If it is, give a proof that it is correct, then describe what data structures and preprocessing you would use to give an efficient version, and give a time analysis.

If it is not correct, give a counter-example showing the strategy is incorrect. Then give a back-tracking algorithm for the problem, and give an upper bound on the time your algorithm would take.

For the last problem, you need to turn in two tables of values. You will need to write and run a program to generate the values, but all you should hand in are the numerical values (and the input sizes they correspond to).

Largest Independent Set for a tree The problem is to find the largest independent set for the special case when the input graph is a tree. (Edges are between nodes and their parents and children.) Remember, an independent set S of a graph is a set of nodes that does not contain both of the endpoints of any edge, i.e. for any edge $\{x, y\}$ either $x \notin S$ or $y \notin S$. So here, we must have a set of nodes of the tree S so that we cannot have both a node and its parent in the set.

Greedy strategy: Find any leaf x in the tree, i.e., any node with no children. Add x to S . Delete x and x 's parent from the tree. Repeat. (Note: when you repeat, the graph might become a *forest*, a collection of unconnected trees, rather than a single tree. In that case, x could be any leaf of any tree. If x does not have a parent, then only it is deleted.)

Intuitively, a leaf in the tree has only one neighbor, its parent. Thus, if we had an independent set not containing the leaf, we can either just add the leaf in, or swap it with its parent. Thus, there is a largest independent set that contains the leaf.

We can make this more precise with a modify the solution proof:

Modify-the-solution lemma: Let F be any forest, and let l be any leaf in F . Let S_2 be an independent set with $l \notin S_2$. Then there is an independent set S_1 with $l \in S_1$ and $|S_1| \geq |S_2|$.

Proof: Let $S_1 = S_2 - \{p(l)\} \cup l$. $|S_1| \geq |S_2| - 1 + 1 = |S_2|$, and $l \in S_1$. The only new element of S_1 is l and $p(l)$, its only neighbor, has been deleted, so S_1 is still an independent set.

It follows: Optimality Lemma: There is a largest independent set in F , S_{opt} with $l \in S_{opt}$.

Proof: There is some largest independent set in F, S' . If $l \in S'$, pick $S_{opt} = S'$ otherwise, we can apply the MtS lemma with $S_2 = S'$ to get S_1 and set $S_{opt} = S_1$.

Then we can prove correctness by strong induction on the size of the forest. We are proving that the greedy strategy produces the largest independent set in any forest T with n nodes. The base case is $n = 1$, i.e. T has a single node l . Then our algorithm produces $S = \{l\}$ which is clearly the largest independent set in T .

Inductively, assume our algorithm produces the largest independent set in any forest with $< n$ nodes. Let F be a forest with n nodes. Then let l be the leaf our algorithm finds in F , and S the independent set our algorithm finds. Let S_{opt} be a largest independent set containing l , which exists by the optimality lemma. By the induction assumption applied to $T - \{l, p(l)\}$, a forest with $< n$ nodes, $S - \{l\}$ is a largest independent set in $T - \{l, p(l)\}$. So in particular, $|S - \{l\}| \geq |S_{opt} - \{l, p(l)\}|$. Now, l in S so $|S - \{l\}| = |S| - 1$, and similarly for S_{opt} . Therefore, $|S| - 1 \geq |S_{opt}| - 1$, and so S is at least as large as S_{opt} , which is the largest possible size.

Thus, by induction, the greedy strategy produces a largest independent set.

We could use a heap to implement the above strategy, but an even more efficiently tuned version is as follows: Keep an array which for each node stores either “In S ”, “Deleted” or “Active”. Keep another array which for each node stores the number of Active children $ChildNo(x)$. Keep a doubly linked list of leaves, i.e. nodes with no active children. Initialize all nodes to “Active”. $ChildNo$ and the linked list can be initialized by doing a DFS of the tree in $O(n)$ time. Then we repeatedly take a node off the list of leaves. We add it to S , and set its array value accordingly. Then we set the value of its parent to “Deleted” if it is not already deleted, and if its parent and grandparent have not been deleted, we decrement the grandparents $ChildNo$. If this goes to 0, we add the grandparent into the leaf list.

This takes $O(1)$ per element added to S , for a total of $O(n)$ time.

Library storage Consider the following problem. A library has n books that must be stored in alphabetical order on adjustable height shelves. Each book has a height and a thickness. The width of the shelf is fixed at W , and the sum of the thicknesses of books on a single shelf must be at most W . The next shelf will be placed on top, at a height equal to the maximum height of a book in the shelf. Give an algorithm that minimizes the total height of shelves used to store all the books. You are given the list of books in alphabetical order, $b_i = (h_i, t_i)$, where h_i is the height and t_i is the thickness.

Greedy strategy: Put books on the first shelf until the total width would exceed W . Then start the next shelf, solving the same problem recursively with the remaining books.

A countering example is the following: Let the books be A,B,C,D,E where A has height 1, width 2, B has height 3 width 4, C has height 3 width 4, D has height 2 width 3 and E has height 2 and width 3, and let $W=8$. Then the greedy strategy puts A and B on the first shelf, C and D on the second, and E on the third, which gives a total height of $\max(1,3)+\max(3,2)+2=8$. A better solution is to put A alone on the first shelf, B and C together on the second, and D and E on the third, to get total height $1+\max(3,3)+\max(2,2)=6$.

A backtracking algorithm is based on repeatedly branching on the possible answers to the question: How many books go on the current shelf? This just returns the total height needed, not the placement, but it would be easy to modify.

BTBookShelves(Books[1..n], W);

1. IF $n = 0$ return 0
2. IF $n = 1$ return $Book[1].height$
3. $J \leftarrow 1$
4. $ThisShelfHeight \leftarrow 0$
5. $ThisShelfWidth \leftarrow 0$
6. $MinPlacement \leftarrow infinity$
7. While $J \leq n$ and $ThisShelfWidth \leq W$ do:
 8. $ThisShelfWidth \leftarrow ThisShelfWidth + Book[J].width$
 9. $ThisShelfHeight \leftarrow \max(ThisShelfHeight, Book[J].height)$
 10. IF $ThisShelfWidth > W$ continue;
 11. $ThisCase \leftarrow ThisShelfHeight + BTBookShelves(Books[J + 1..n], W)$.
 12. {If we end the shelf at book J , we need to add the current shelf's height to the best placement for the rest of the shelves}
 13. IF $ThisCase < MinPlacement$ THEN $MinPlacement \leftarrow ThisCase$.
14. $J ++$
15. Return $MinPlacement$

Since each recursive call is to a smaller array, the recursion depth is at most n , and since the number of recursive calls is at most n , this gives a crude upper bound for the time by n^n . However, we can get a tighter analysis as

follows: On input n , the first recursive call is to size $n - 1$, the second to size $n - 2$, and so on. Thus, $T(n) \leq T(n - 1) + T(n - 2) + \dots T(1)$. We can then prove by strong induction that $T(n) \leq c2^n$ for some c , since assuming this is true for smaller values of n , $T(n) \leq T(n - 1) + T(n - 2) + \dots T(1) \leq c(2^{n-1} + 2^{n-2} + \dots 1) = c(2^n - 1) < c2^n$. Thus, $T(n) \in O(2^n)$.

See the next assignment for a DP version.

Halloween Route Consider the following problem. The input is a positive real number T and a labelled graph whose nodes represent houses in a neighborhood. Each house is labelled with the number of pieces of candy that the residents give to trick-or-treaters. The edges represent paths between the houses. Each edge is labelled with the time it takes to walk down the path in your Halloween costume. Also given as part of the input is the total time T you can stand to be in your hot uncomfortable costume. Your goal is to find a route through the graph starting at your house (but it doesn't have to end there) that maximizes the amount of candy you get. You can go by the same house more than once, but you only get candy the first time.

Candidate greedy strategy: Until you run out of time do: from the current house, go to the neighbor that maximizes the ratio of candy available at the house and the distance to the house (either directly or indirectly, through other houses, from the current house). Treat visited houses as having 0 candy.

Consider five houses: You start at house A, house B is two hours away and has 5 pieces of candy, houses C, D, and E are one hour away and have 2 pieces each, and there is a path from C to D and E and a path from D to E, each of distance one hour. You have 4 hours for trick or treating. The ratios for the houses are 2.5 for B and 2 for each of the others, so the greedy schedule would involve going to B. Since by the time you get back, Halloween is over, the greedy strategy nets you 5 pieces total. On the other hand, going to C, D, and E takes 3 hours total, and gets you 6 pieces. So the greedy strategy is sub-optimal on this counter-example.

The following backtracking algorithm assumes that all houses are reachable from each other. If this is not case, for each non-edge, insert an edge with infinite cost. We'll use $d(u, v)$ to represent the distance from house u to house v . At each step, we'll branch on which house to consider next.

Halloween(V : houses, Candy[V]: candy at houses, $d[V][V]$:distances between houses, T :time remaining, s : starting house);

1. IF $T = 0$ return $Candy[s]$.
2. IF $n = 1$ return $Candy[s]$
3. FOR each $u \in V - \{s\}$ do:

4. FOR each $w \in V - \{s, u\}$ do:
5. $d[u][v] \leftarrow \min(d[u][v], d[u][s] + d[v][s])$ {after we delete s , we will still consider shortcuts through s as a way to go between other houses}
6. $BestCandy \leftarrow Candy[s]$. {We get this amount to start}
7. For each $u \in V - s$ do:
8. IF $d[s][u] \leq T$ do:
9. $FutureCandy \leftarrow Holloween(V - \{s\}, Candy[V - \{s\}], d[V - \{s\}][V - \{s\}], T - d[s][u], u)$ { If we go to u next, how much candy can we get?}
10. IF $Candy[s] + FutureCandy > BestCandy$ THEN $BestCandy \leftarrow Candy[s] + FutureCandy$.
11. Return $BestCandy$

Since we delete our current house from future calls, we make at most $n - 1$ calls at the top level, $n - 2$ at each of the second levels down, and so on, to get a time that's at most $O(n!)$. However, in many cases we can get a better bound. Let $M = T / \min_{u,v} d[u][v]$. We can go at most M steps before running out of time, so we can bound the total time by $O((n - 1) \dots (n - M)) = O(n! / (n - M - 1)!) < O(n^M)$.

Oxen pairing Consider the following problem: We have n oxen, Ox_1, \dots, Ox_n , each with a strength rating S_i . We need to pair the oxen up into teams to pull a plow; if Ox_i and Ox_j are in a team, we must have $S_i + S_j \geq P$, where P is the weight of a plow. Each ox can only be in at most one team. Each team has exactly two oxen. We want to maximize the number of teams.

Candidate Greedy Strategy: Take the strongest and weakest oxen. If together they meet the strength requirement, make them a team. Recursively find the most teams among the remaining oxen.

Otherwise, delete the weakest ox. Recursively find the most teams among the remaining oxen.

This strategy is optimal. We'll prove it using the following two lemmas:

Lemma 1: Let s be the strongest ox, and w the weakest. If $s + w < P$, then there is no set of teams that assigns w to any team.

Proof: If $Teams$ is a set of teams in which w is assigned to a team with some ox s' , since $s' \leq s$, $s' + w \leq s + w < P$, the team could not actually pull the plow. This contradiction proves the lemma.

Lemma 2: Let s be the strongest ox, and w the weakest. Assume $s + w \geq P$. Let $Teams_2$ be a set of disjoint teams that can all pull the plow does

not pair s with w . Then there is a set of disjoint teams $Teams_1$ that can all pull the plow, which assigns w to a team with s and is so that $|teams_1| \geq |Teams_2|$.

Proof: If $Teams_2$ does not assign both s and w to teams, let $Teams_1$ be $Teams_2$ less any team that includes s or w , together with the team (s, w) . Since $s + w \geq P$, and (w, s) is the only team we added, this is a set of disjoint teams that can all pull the plow. Since at most one of s, w were in a team we deleted at most one team, and added one team, so $|Teams_1| \geq |teams_2|$.

Otherwise, let $Teams_2$ partner s with x and w with y . Let $Teams_1 = Teams_2 - \{(s, x), (w, y)\} \cup \{(x, y), (w, s)\}$. $Teams_1$ is a set of disjoint teams, and (w, s) can pull the plow by our assumption. Now, since w and y can pull the plow, and x is at least as strong as the weakest ox w , x and y can pull the plow. Thus, all teams that we added can pull the plow. Since we deleted two teams and added two teams, $|teams_1| = |teams_2|$. Thus, we have proved the lemma in both cases.

We can now prove that the greedy strategy is optimal:

Theorem: There is so set of legal teams $Teams_2$ greater than that produced by the greedy strategy.

We prove this by strong induction on n , the number of oxen. Assume that the greedy strategy is optimal on all sets of size $< n$. Then on a set $Oxen$ of size n , let w and s be the weakest and strongest oxen. Assume $Teams_2$ is larger than the greedy strategy's solution $Greedyteams$. If $w + s < T$, then by lemma 1, neither $Teams_2$ nor $Greedyteams$ contains a team with w . Thus, by the induction hypothesis, $Greedyteams$ is the best solution for $Oxen - \{w\}$, and $Teams_2$ is some solution for $Oxen - \{w\}$, so $|Teams_2| \leq |Greedyteams|$.

If $w + s \geq T$, then by lemma 2, there is a solution $Teams_1$ which like the greedy solution, pairs (w, s) and $|Teams_1| \geq |Teams_2|$. Then, since $GreedyTeams - \{(w, s)\}$ is an optimal solution for $Oxen - \{s, w\}$ by the induction hypothesis, and $Teams_1 - \{(w, s)\}$ is a legal solution for this problem, $|Teams_1 - \{(w, s)\}| \leq |GreedyTeams - \{(w, s)\}|$ so $|Teams_1| - 1 \leq |GreedyTeams| - 1$, so $|Teams_2| \leq |Teams_1| \leq |GreedyTeams|$. So the greedy solution also produces optimal teams on a set of size n .

By induction, the greedy solution is optimal for all N .

To get an efficient version of the algorithm, first sort the oxen by strength. We either delete the weakest or both the weakest and strongest, so the set that is left is of the form $Oxen[i..j]$. We just need to keep track of i and j . The following algorithm, after the input is sorted, does so:

1. $Teams \leftarrow \emptyset$

2. $I \leftarrow 1, J \leftarrow n$.
3. While $I < J$ do:
 4. IF $Oxen[I] + Oxen[J] \geq T$ THEN $Teams \leftarrow Teams \cup \{(I, J)\}$,
 $I ++, J --$.
 5. ELSE $I ++$.
6. Return $Teams$.

Since $J - I$ always decreases by at least one, the above loop executes at most $n - 1$ times, so the above loop takes $O(n)$ time. However, we need to spend $O(n \log n)$ time to sort the inputs, which gives $O(n \log n)$ total time.

Implementation of Independent Set Consider a greedy heuristic for independent set that selects the lowest degree node, puts it in the set, and deletes it and its neighbors and repeats. Implement this greedy heuristic, and a correct backtracking algorithm for Maximum Independent Set. Run both algorithms on random graphs where the graphs are constructed by adding edges between each pair of nodes independently with probability $1/2$. For various input sizes n , plot the average sizes of the greedy heuristic sets compared to the optimal independent sets found by the backtracking algorithm. You should try to get data for n as large as possible, and plot the results on a $\log n$ vs. size found (note: not \log size found) graph. Can you give a conjecture as to how the independent set grows as a function of size? (Don't hand in code, but describe your algorithm, programming language, and architecture. Your grade for this problem will be partially based on how large n you can get your backtracking algorithm to run on.)

For large enough n , the largest independent set in a graph approaches $2 \log n$, while the greedy algorithm finds sets of size approximately $\log n$. However, for small n , you are unlikely to see sets that large. Since the actual independent set is small, the backtracking independent set algorithm will typically take time $n^{\log n}$ rather than 2^{6n} . However, for $n = 2^{10}$, this is still $(2^{10})^{10} = 2^{100}$ so will be prohibitive unless you use some tricks to speed up the computation.