The Greedy Approach as an Algorithm Design Technique

Now we want to study an algorithm design technique that sometimes works, namely the "greedy approach." The idea of the greedy approach is to solve an instance of a problem, typically an optimization problem, by making the *locally optimal choice* whenever a decision has to be made.

> For the brute force technique, divide and conquer, and dynamic programming, the *correctness* of the resulting algorithms was fairly obvious. For the greedy approach, proving the correctness of the algorithm is the hard part. It is pretty easy to design an algorithm for a problem based on the greedy approach, but it is often not really an algorithm, because it doesn't always work!

In this Chapter we will very briefly study the greedy approach. The only Exercise, and corresponding possible Test 2 question, is to prove that a proposed greedy algorithm for a problem is not correct by providing a counter-example—an instance of the problem where the greedy algorithm fails to produce the correct answer.

After some introductory work on this topic, three famous greedy algorithms will be presented, and it is hoped that you will read over this material and watch the videos, but there are no Exercises or Test 2 questions from this later material in the chapter.

The Change-Making Problem and Failure of the Greedy Approach

The change-making problem is to find the smallest number of coins possible that make up a given amount, where we have any number of coins of every possible denomination, and the coins are of denominations $1 = d_1 < d_2 < \ldots < d_n$.

What is the greedy approach to the change-making problem? Well, since the objective is to make the target amount of money with as few coins as possible, one reasonable, greedy approach would be to use the largest denomination coin first, and then, when we can't use another of this denomination, switch to the next denomination, and so on.

For example, if $d_1 = 1$, $d_2 = 5$, $d_3 = 10$, and $d_4 = 25$ (standard U.S. coins), the greedy approach to making 71 cents would be to use two quarters, leaving 21 cents, then use two dimes, and then use a single penny.

It turns out (though we won't prove it) that the greedy approach does work for these denominations of coins.

On the other hand, if we have $d_1 = 1$, $d_2 = 5$, and $d_3 = 6$, then the greedy approach to making 10 cents would be to first use a 6 cent coin, leaving 4 cents, and then be unable to use a 5 cent coin, and so have to use four 1 cent coins, leaving us with five coins used. Of course, using two 5 cent coins would produce the target amount, with a lot fewer coins. This example proves that the greedy approach fails for these denominations of coins and this amount. Hence, the greedy approach does not produce a generally correct algorithm for the change-making problem.

Showing that a Greedy Algorithm for 0-1 Knapsack Can Fail

Consider the 0-1 knapsack problem. A greedy approach is to sort the items into most valueable per unit weight order, and then to simply use each item, in order, that fits into the remaining capacity of the knapsack.

Let's try to come up with an example for which this alleged algorithm fails. We want our example to be as simple as possible.

Clearly if there is only one item, there are only two possible subsets—the empty set (the answer if the item is too heavy to fit in the knapsack all by itself), and the set consisting of the single item (the answer if the item fits).

Next, if we have two items, namely item 1 with profit p_1 and weight w_1, and item 2 with profit p_2 and weight w_2, and they are sorted so that $\frac{p_1}{w_1} > \frac{p_2}{w_2}$, then the greedy approach is to use item 1 if it fits, which will be better than using item 2, even if item 2 fits, because item 1 is more valuable per unit of weight. If they both fit, the correct solution is to use them both, which is what the greedy approach does. The point is, there doesn't seem to be an example with just two items where the greedy algorithm fails.

So, we might now be wondering if the greedy approach always works. But, for three items, we can easily concoct an instance where using the first item is a mistake. For example, consider these three items with total knapsack capacity of 10:

Item i	p_i	w_i	$\frac{p_i}{w_i}$
1	72	6	12
2	55	5	11
3	50	5	10

Clearly the greedy approach will choose to use item 1, for a profit of 72 and weight used up of 6, and then there won't be room in the knapsack for either item 2 or item 3. This is not optimal because both items 2 and 3 can fit snugly in the knapsack, producing a total profit of 105.

Greedy Approach on ETSP

As another example of considering whether the greedy approach gives a correct algorithm, consider the Euclidean Traveling Salesperson Problem (ETSP).

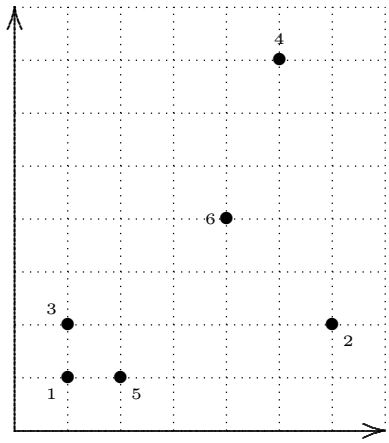
Given n vertices $v_j = (x_j, y_j)$ in the plane, ETSP is to find the tour of these vertices (starting with v_1 as we did for the TSP in the dynamic programming chapter) that has minimal total Euclidean length, where the length of an edge between v_j and v_k is

$$\sqrt{(x_j - x_k)^2 + (y_j - y_k)^2}.$$

Thus, ETSP is simply a specialized version of TSP where the weights of the edges are the ordinary geometric length of the edges.

Here is an example instance of ETSP:

$v_1 = (10, 10)$, $v_2 = (60, 20)$, $v_3 = (10, 20)$, $v_4 = (50, 70)$, $v_5 = (20, 10)$, $v_6 = (40, 40)$
Here is a graph of these points (where the grid lines are at multiples of 10):



Here are the weights (distances) for this problem, rounded to 2 decimal places:

	1	2	3	4	5	6
1	0	50.99	10	72.11	10	42.43
2	50.99	0	50	50.99	41.23	28.28
3	10	50	0	64.03	14.14	36.06
4	72.11	50.99	64.03	0	67.08	31.62
5	10	41.23	14.14	67.08	0	36.06
6	42.43	28.28	36.06	31.62	36.06	0

Now consider a tour, say 1—5—6—2—4—3—1. The total weight of this tour is

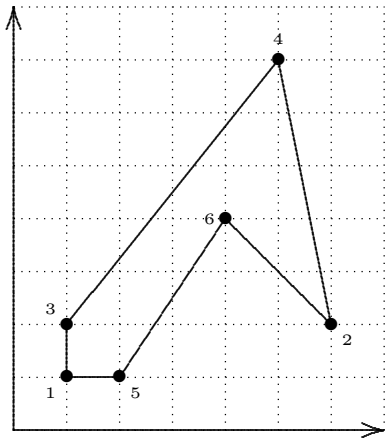
$$10 + 36.06 + 28.28 + 50.99 + 64.03 + 10 = 199.36.$$

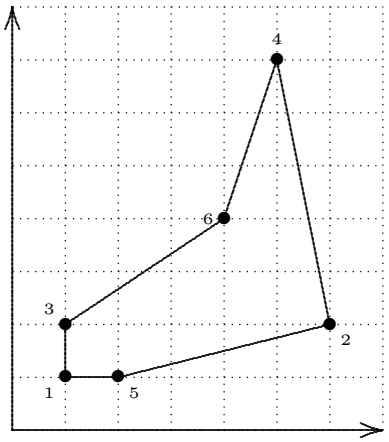
Is this the optimal tour? No, it is not, because the tour 1—5—2—6—4—3—1 has total weight

$$10 + 41.23 + 28.28 + 31.62 + 64.03 + 10 = 185.16,$$

which is better than the first tour.

Graphing tours can help us to picture what is going on (but the difference in total weight between these two tours isn't so obvious):



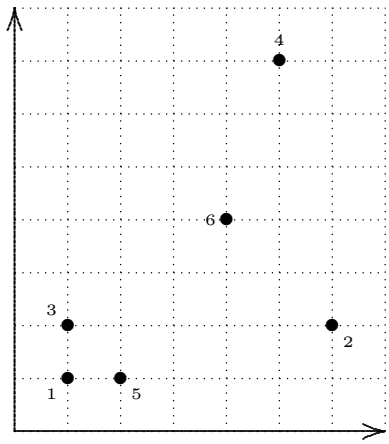


Exercise 24 [10 points] (target due date: Monday, October 19)

Consider the following algorithm for ETSP, using the greedy approach:

> start with vertex 1 as the current vertex (but don't mark it used)
>
> repeat:
>
> > find the unused vertex that is closest to the current vertex
> > (if there is more than one unused vertex tied for closest, use the one that has the smallest index, just to be definite),
> >
> > add to the tour the edge from the current vertex to this vertex,
> >
> > mark the added vertex used, and
> >
> > make it the current vertex
>
> until there are no more unused vertices

Your first job on this Exercise is to demonstrate this proposed algorithm on the previous instance of ETSP, sketching the tour here:



Write the length of each edge in the tour next to that edge, and c ompute the total length of this tour, and show that this is worse than some other tour's total length.

Note that this instance is an example of size 6 showing that the proposed greedy algorithm is not always correct.

Your second job on this Exercise is to find the simplest (smallest n) example you can showing that the proposed greedy algorithm can fail. Show what tour is found by the greedy algorithm for your example, write the length of each edge on your picture, compute the total length of the tour, and show that this is not the optimal tour by doing the same picture for a better tour.

Note: you should look for an example with $n = 2$, $n = 3$, and so on, up to and including $n = 5$ (if you decide there are no counterexamples of size less than 6, then you should conclude that the previous example is what you are looking for), hoping to find the smallest possible counterexample to the claim that this greedy algorithm always works. You will not be given credit for this Exercise unless you correctly find the smallest possible counterexample.

You **do not** have to use the previous $n = 6$ example in your quest for the smallest counterexample—it is a counterexample of size 6, and has nothing to do with your search. You may draw any points you wish in constructing, out of nothing, your possible counterexamples.

The "Find a Minimum Spanning Tree" Problem

Now we will study two classical algorithms for solving the problem of finding a minimum weight spanning tree for a given graph, or showing that the graph is not connected. We are of course studying these algorithms now because they were invented by using the greedy approach.

Consider a weighted, undirected graph (this just means that the weight matrix is symmetric, so that we can draw a single edge with no arrows between two vertices instead of drawing arrows both ways with the same weight on them). The problem we want to tackle now is: find a tree (collection of edges with no cycles) whose edges are a subset of the edges in the given graph that spans the graph and has minimum total weight among all such spanning trees, or realize that the graph is not connected.

When we say a set of edges "spans the graph" we mean that for any two vertices in the graph, there is a path between them that only uses edges in the spanning set.

We say that a graph is "connected" (nothing to do with organized crime) if its edges span it.

A General Psuedo-Algorithm for Finding a MST

Here is a general framework for finding a minimum spanning tree (MST) by a greedy approach. We will see that Prim's and Kruskal's algorithms both fit into this framework.

In this psuedo-code (like in Python, indentation indicates block structure), F is a set of edges that at the end is the MST (unless it does not span all the vertices in the graph, in which case the graph is unconnected).

```
F = φ
While( F does not form a spanning tree )
    Select an edge by some greedy (locally optimal) method
    If( adding that edge doesn't make F have a cycle )
        Add the edge to F
```

Prim's Algorithm

In addition to maintaining F, Prim's algorithm maintains a set Y of vertices that are currently reachable from v_1 using edges currently in F. Y is initialized to be $\{v_1\}$.

Prim's algorithm does the "select an edge" part of the framework by selecting a vertex v in $V - Y$ (where V is the set of all the vertices in the graph) that is nearest to a vertex y in Y. Then it adds v to F and adds the edge between v and y to Y.

If it ever encounters a situation where there is no vertex reachable from a vertex in Y, it halts and reports that the graph is not connected.

On a connected graph, the algorithm stops, knowing that F is a spanning tree, when $Y = V$.

⇒ Why doesn't Prim's algorithm need to explicitly check whether the new edge being added to F creates a cycle as suggested in the framework?

⇒ Use these weights in an undirected graph and trace the execution of Prim's algorithm:

	1	2	3	4	5	6
1	–	7	11	17	2	9
2	–	–	15	18	11	3
3	–	–	–	16	2	10
4	–	–	–	–	14	5
5	–	–	–	–	–	2
6	–	–	–	–	–	–

While doing this, develop a scheme that could actually be implemented using arrays `nearest` and `distance` where

`nearest[j] =` the index of the vertex in Y that is nearest to v_j
`distance[j] =` the weight of the edge between v_j and the vertex with index `nearest[j]`

Each time a vertex v is added to Y, these arrays need to be updated.

⇒ Do efficiency analysis of Prim's algorithm, paying close attention to how much work it takes to keep `nearest` and `distance` updated.

Proof of the Correctness of Prim's Algorithm

As we said earlier, the greedy approach requires a careful proof of correctness.

Define a set of edges to be "promising" if it is a subset of some MST. We will only work with the situation that the graph is connected.

Suppose F is a set of edges that is promising and Y is the set of vertices on the ends of the edges in F, and suppose that e is an edge of minimum weight connecting a vertex y in Y, to a vertex v, in $V - Y$ (all of this setup, except for the assumption that F is promising, is simply the situation in a snapshot of Prim's algorithm).

⇒ Prove that $F \cup \{e\}$ is promising by letting $F' \supseteq F$ be a MST and consider the cases that $e \in F'$ and $e \notin F'$. In the first case it is trivial to see that $F \cup \{e\}$ is promising, and in the second case, since F' is spanning, there is a path in F' connecting y and v that does not use e, so there is some other edge e' on the path from y to v that does not pass through e, and we can show that $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$ which is an MST.

⇒ Use the previous fact, and induction on the number of elements in Y, to prove that Prim's algorithm is correct.

Kruskal's Algorithm

In addition to maintaining F, Kruskal's algorithm maintains a collection of subsets of V that partition V (any two of these subsets don't overlap and the union of all of them is is V). In addition to setting $F = \phi$, Kruskal's algorithm initializes this collection of subsets to be $\{v_1\}$, $\{v_2\}$, ..., $\{v_n\}$.

Also, Kruskal's algorithm pre-sorts the edges E of the given graph into non-decreasing weight order.

For the step in the framework that says "select an edge" Kruskal's algorithm simply selects the next edge in the sorted order. Then it checks to see if the edge connects two vertices that are in the same subset. If it does, the edge is *not* added to F. If it connects two vertices in different subsets, then it is added to F, and the set of subsets is updated to combine those two subsets into one subset.

Kruskal's algorithm halts after examining the last edge. If there is only one subset remaining, then F is the MST. If there is more than one subset remaining, then the graph is not connected.

Kruskal's algorithm requires an efficient data structure to implement a collection of disjoint subsets. The operations we need in this data structure are "given a vertex, find the subset it belongs to," "given two subsets, merge them," and "initialize the subsets."

One way to do this is to use a sort of upside-down tree structure, where each vertex is stored in a node that points to another node in the same subset, with the root of the tree

pointing to itself. At the beginning we make a node for each of 1 through n, each pointing to itself, and when we merge two trees, we make the root of the shorter tree point to the root of the taller tree, instead of to itself. When we are given an arbitrary vertex, we go to its node (we have to have pointers to the nodes from the vertices) and follow the pointers until we reach the root node (recognized because it points to itself), and use that root node's vertex as the representative element of the subset. So, two vertices end up pointing to the same root node if and only if they are in the same subset.

⇒ Use the weights given previously and trace the execution of Kruskal's algorithm, using this pictorial representation of the disjoint sets data structure.

It is fairly easy to implement this "disjoint sets data structure" using arrays `root` and `depth` where

`root[j] =` the index of the representative vertex for the subset v_j belongs to

`depth[j] =` the depth of the tree structure if j is the representative vertex for its subset.

⇒ Do efficiency analysis of Kruskal's algorithm, paying close attention to how much work it takes to keep `root` and `depth` updated.

Proof of the Correctness of Kruskal's Algorithm:

Given the situation and notation as in Kruskal's algorithm, and using our earlier definition of "promising," let F be promising and let e be an edge of minimum weight in $E - F$ that that $F \cup \{e\}$ has no cycles. Then we claim that $F \cup \{e\}$ is promising.

⇒ Follow much the same reasoning as in the proof of correctness of Prim's algorithm to prove this.

⇒ Use induction on the number of edges in F and this previous fact to show that Kruskal's algorithm produces an MST.

Huffman's Algorithm

Consider the problem of creating an efficient prefix code for a given frequency distribution of symbols. Huffman figured out how a greedy approach could be used to construct an optimal prefix code for a given frequency distribution of a set of symbols.

A prefix code is a mapping between a set of symbols and a set of sequences of 0's and 1's, where no code for a symbol is an initial substring of any other code. This means that you can go through a sequence of bits and extract the corresponding symbols without confusion.

A fixed-length code is a common approach to this problem. If we have N distinct symbols, we can use roughly $\log N$ bits to represent each. For example, in the old-time ASCII code 7 bits were used to represent a single symbol, yielding the possibility of $2^7 = 128$ different symbols. In this code, `A` was represented by `1000001` (65 in decimal).

With a fixed-length code, using m bits per symbol, the number of bits required to represent any message of length L is simply mL.

It turns out that we can take advantage of varying frequencies of symbols in meaningful text to represent L symbols in a smaller number of bits.

A binary tree where each node has 0, 1, or 2 children, with the left branch representing 0 and the right branch representing 1, corresponds to a prefix code in an obvious way: the leaves are the symbols, and the path from the root to each leaf gives the sequence of bits that encode that symbol. Given the tree, it is easy to scan a sequence of bits and produce the corresponding symbols.

⇒ Draw a randomish tree of this kind and assign letters to the leaves. Note how the tree can be used to encode and decode.

We can now state our actual problem: given a list of all the symbols we want to encode, with each having a percentage of occurrence in the source document (actually we can use any number assigned to each symbol such that the larger the number the more likely the symbol is to appear), construct a tree that gives a code that has a minimum number of bits required to encode the source document.

We will present the algorithm through an example.

Suppose we scan a source document and find these symbols and percentages of occurrence (these are actual counts from a bunch of sources):

Symbol	Percentage	Symbol	Percentage
E	12.51	T	9.25
A	8.04	O	7.60
I	7.26	N	7.09
S	6.54	R	6.12
H	5.49	L	4.14
D	3.99	C	3.06
U	2.71	M	2.53
F	2.30	P	2.00
G	1.96	W	1.92
Y	1.73	B	1.54
V	0.99	K	0.67
X	0.19	J	0.16
Q	0.11	Z	0.09

The greedy approach idea of Huffman's algorithm is to make a forest of single-node trees, one for each symbol, and then to repeatedly find the two lowest frequency trees in the forest and combine them into one tree with its root having the frequencies of the two trees added together. When only one tree remains, it is the desired Huffman tree giving the optimal code for the given frequency distribution.

⇒ Create the Huffman tree for this chart and use it to create a prefix code. Note how many bits it would take to represent typical English text using a fixed-length code versus this code.

⇒ Ponder the efficiency of this algorithm. We could use a priority queue data structure, based on the heap data structure, to maintain the forest of trees with the ability to efficiently get the lowest frequency tree. Would it be worth it?

Note that we are not proving the correctness of this algorithm—Huffman did that, according to Wikipedia, as a final class project in a graduate course at MIT, hoping to avoid taking a final exam, and subsequently published a journal paper on this work in 1952.
