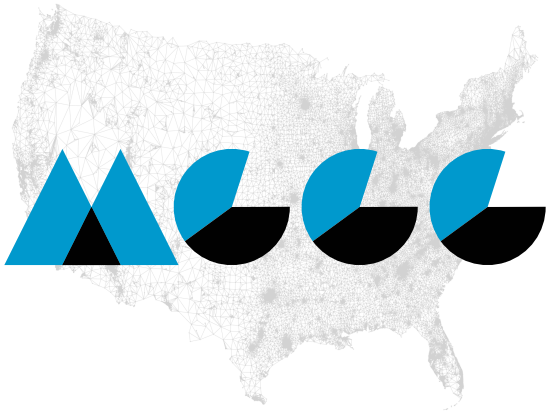
Optimization Breakouts



Zachary Schutzman, Sarah Cannon, Austin Buchanan, Justin Solomon

A brief, hands-on introduction to four topics in optimization.

Contents

1 Day 1: *Greedy Algorithms*

Greedy algorithms are an important class of algorithms used to solve computational problems. Similar to 'divide-and-conquer' or 'dynamic programming', 'greedy' refers to the general style of the process rather than a specific algorithm.

Definition 1 (Greedy Algorithm)**.** A **greedy algorithm** is an algorithm in which we find a *globally optimal* solution to some problem by iteratively selecting the *locally optimal* option at each step.

We use greedy algorithms all the time without even realizing it. When you walk somewhere, at each intersection, you choose to walk down the street which points toward your destination. You can use the following algorithm to make change using the fewest number of coins with US denominations ($0.25, $0.10, $0.05, $0.01):

1.1 An Example

Algorithm 1 Greedy algorithm for making change

Input: An amount x (in cents) of money to give as change.
Output: The smallest collection C of coins needed to make that amount.

```
C ← {}.                                          ▷ Initialize the output.
while x > 0 do
    Add to C the largest denomination coin b with value less than or equal to x.
    x ← x − b                                    ▷ Update x
return C.
```

Exercise 1. *Implement the above algorithm in Python, using the standard United States coinage values ($0.25, $0.10, $0.05, $0.01).*

Since greedy algorithms are so intuitive, they are often very easy to construct, analyze, and verify. Let's do a quick proof of correctness here. To show the correctness of an algorithm we need to prove that for any input, the output the algorithm gives us satisfies the things we claim it should. Here, we need to prove that the algorithm always outputs a collection of money with total value x and that there doesn't exist a smaller collection with the same value.

Claim 1. *The algorithm above for making change always outputs a collection C such that the total value of the bills and coins in C is exactly equal to x and there does not exist a collection C' with total value x and strictly fewer coins than C.*

Proof. We'll start by showing that the algorithm always outputs an amount of money equal to x. First, observe that the WHILE loop runs as long as the current value of C is less than x, so it will certainly never output a C with value that is too small.

If the algorithm were to output something with too large of a value, it must have been the case that we entered the WHILE loop with x set to some value and then we added a denomination strictly larger than x, but this is not allowed, since we can only add to C something with value less than or equal to x, so we never output a C with too much money in it.

We finally need to show that the WHILE loop actually terminates. To do this, we can observe that since any amount of money can be created with a large enough pile of pennies, that adding $\$.01$ to C will always be an option, so the algorithm never gets stuck. In these three pieces, we have that the algorithm always terminates and outputs a collection of coins C with value exactly equal to x.

The optimality of this algorithm is very easy to check by hand. You can prove it fully by induction but there are only 99 cases. You can use a bit of induction to reduce it to 24 cases by observing that for any x greater than $\$0.25$, the correct thing to do (which is what the algorithm does) is add a quarter to C, which reduces x by $\$0.25$.

Next, consider the four denominations as defining intervals, and our value x falls into exactly one of these, i.e. x is either between $\$0.01$ and $\$0.05$, $\$0.05$ and $\$0.10$, $\$0.10$ and $\$0.25$, or greater than $\$0.25$. We'll argue that in each of these cases, the optimal solution must include the coin which has value equal to the lower bound of the interval containing x, which is what the greedy algorithm does.

Call this largest available coin c_k and suppose, for the sake of contradiction, that the optimal solution does not include c_k but the greedy solution does. Then the optimal solution must use only coins smaller than c_k to sum to the value x, but it's straightforward to check that this is not possible, since no smallest collection can include more than four pennies (use a nickel instead), more than two nickels, or more than two dimes, so this replacement is impossible and the greedy algorithm does indeed find the optimal solution.

□

That this algorithm works is (sadly) particularly dependent on the allowed denominations. For example, if we had coins with values $\$0.01$, $\$0.07$, and $\$0.10$, then the optimal way to make $x = \$0.14$ is to use two $\$0.07$ coins, but the greedy algorithm will pick one $\$0.10$ coin and four $\$0.01$ coins.

Exercise 2. *What conditions are required of a collection of denominations such that the greedy algorithm finds the smallest collection of coins for any input value?*

1.2 Minimum Spanning Trees

Another setting where greedy algorithms have a lot to say is the **minimum spanning tree** problem. First, some notation and definitions. Throughout, all graphs are *simple*, meaning that they never have more than one edge between a pair of vertices and no vertex has an edge to itself (called a 'self-loop'), and *connected*, meaning that there exists at least one path between every pair of vertices.

A (weighted) graph G has a vertex set V and edge set E, consisting of pairs of adjacent vertices, which we write as $G = \langle V, E \rangle$. We write (u, v) for the edge connecting vertices u and v and w_{uv} denotes the *weight* on edge (u, v). Weighted graphs are very useful abstractions. We can think of a transportation network as a weighted graph where the vertices are cities, the edges are roads, and the weights of an edge is the travel time on that road. In a social graph, the vertices can represent

individuals, an edge between vertices exists if the two individuals are friends, and the weight can represent the strength of the relationship.

A **tree** is a special kind of graph, and like many mathematical objects, has several equivalent definitions.

Definition 2. A **tree** T is a graph satisfying any one of the following equivalent conditions:

1. T has no cycles
2. Between any two vertices x and y in T, there is a unique path joining x and y
3. T has exactly one fewer edges than vertices
4. T is such that removing any edge disconnects it

Exercise 3. *Prove that these four conditions are indeed equivalent.*

Feel free to work with whichever is the most convenient for your problem.

Definition 3. Given a connected graph G, a **spanning tree** of $G = \langle V, E \rangle$ is a subgraph H of G such that

1. H contains all of the vertices of G
2. H is a tree

Spanning trees have important applications in settings like communication networks, where cycles in the graph can lead to nodes getting redundant information, or, worse, endlessly passing packets around a cycle.

For any given graph, finding some spanning tree is a pretty simple task, and probably your first guess at a procedure for constructing one is a correct algorithm. It's also probably a greedy algorithm. Here are two ideas for how you might do this.

1. Pick an arbitrary vertex in the graph and add it to the tree. Iteratively choose an arbitrary vertex not in the tree which is adjacent to one that is, and add it to the tree along with an edge joining it to the rest of the tree.
2. Remove all the edges from the graph and arbitrarily order them. Iteratively try to add the first one in the list to the graph. If adding the edge creates a cycle, leave it out and move on to the next one. Otherwise, add it and move on to the next one.

We can see that both of these procedures do indeed create a spanning tree, since the first one creates a subgraph satisfying definition (3) of a tree and the second one creates a subgraph satisfying definition (1) of a tree. Both of these algorthms are *greedy* in that they only need to make a decision about a single vertex/edge at each step. We'll use these as foundations for the problem we're really interested in: finding a *minimum spanning tree*.

Given a graph G with edge weights, we might be interested in finding the spanning tree with minimum weight. That is, given any spanning tree of G, we can look at the total weight of the edges in the tree, and we want to find the one with the smallest possible weight. This is again important

in communication networks where the weight of each edge represents the cost to transmit the information between those two vertices. If we want to spread information in the network at the least cost, the minimum spanning tree tells us which edges we should use.

Let's adapt the two procedures above to find minimum spanning trees. The first extends to *Prim's Algorithm*:

Algorithm 2 Prim's Minimum Spanning Tree Algorithm

Input: A graph $G = \langle V, E \rangle$ with edge weights w_{uv}.
Output: A minimum spanning tree $T = \langle V_T, E_T \rangle$ of G.

$T_V \leftarrow \{x\}$, $E_T \leftarrow \{\}$. ▷ Add an arbitrary vertex x to the tree
while $|E_T| < |V| - 1$ **do** ▷ While T is not yet a spanning tree
    Add to T_E the minimum-weight edge (u, v) such that $u \in T_V$, $v \notin T_V$, add v to T_V
return $T = \langle T_V, T_E \rangle$.

By the same argument as earlier, this algorithm finds a spanning tree of G, but we need to argue that it finds a *minimum* spanning tree.

Claim 2. *The spanning tree found by Prim's Algorithm has weight no larger than any other spanning tree.*

Proof. Suppose, for the sake of contradiction, that Prim's Algorithm finds a spanning tree T but there is another spanning tree S with strictly smaller weight. We know that the set of edges in S must not be the same as that of T, so consider the first time at which the algorithm adds an edge to T which is not in S. Call this edge (x, y) and let x denote the end of the edge which was already in the partially-formed T. Since (x, y) is in T and not in S, there must be some other path in S joining x and y which does not use edge (x, y), and let $x = p_0, p_1, p_2, p_3, \ldots, p_k = y$ be the vertices along this path. Since, at this moment, x is in T and y is not, there must be some edge (p_i, p_j) such that p_i is in T and p_j is not.

Since Prim's Algorithm decided to add edge (x, y) and not edge (p_i, p_j), it must be that the weight w_{xy} of edge (x, y) is less than or equal to the weight $w_{p_i p_j}$ of edge (p_i, p_j). Call S' the graph we get by deleting from S the edge (p_i, p_j) and instead including edge (x, y). This graph is still connected and has the same number of edges as S, so it is a spanning tree. Additionally, since $w_{xy} \leq w_{p_i p_j}$, the total weight of the edges in S' is not larger than that of S, so it is also a minimum spanning tree. Additionally it contains all of the same edges as T up to and including the addition of edge (x, y), so there is a later 'first time' at which Prim's algorithm adds to T an edge not in S'.

Repeating this argument allows us to conclude that all of the edges Prim's Algorithm adds to T are present in a minimum spanning tree, so T is itself a mimimum spanning tree. □

The second procedure described earlier extends to *Kruskal's Algorithm*.

Again, the same argument as before shows that Kruskal's Algorithm finds a spanning tree, and we need to show that this is indeed a minimum spanning tree.

Claim 3. *The spanning tree found by Kruskal's Algorithm has weight no larger than any other spanning tree.*

Exercise 4. *Prove this claim. As a hint, proceed inductively. Kruskal's Algorithm starts with the edge with the smallest weight and iteratively adds the edge with the smallest weight which does not form a*

Algorithm 3 Kruskal's Minimum Spanning Tree Algorithm

Input: A graph $G = \langle V, E \rangle$ with edge weights w_{uv}.
Output: A minimum spanning tree $T = \langle V_T, E_T \rangle$ of G.

```
T_V ← V, E_T ← {}.                          ▷ Start with no edges
Sort E by weight, smallest to largest
while |E_T| < |V| − 1 do                    ▷ While T is not yet a spanning tree
    e ← pop(E)                              ▷ Consider the first edge in E
    if E_T ∪ {e} is still a tree then
        Add e to T_e
    elseDiscard e
return T = ⟨T_V, T_E⟩.
```

cycle. The base case is that this smallest-weight edge is a member of every minimum spanning tree and the inductive hypothesis is that, given some collection of edges which belong to the minimum spanning tree, a smallest-weight edge which, when added, does not form a cycle is also a part of a minimum spanning tree.

Exercise 5. *Implement Kruskal's Algorithm in Python. A recommended approach is to have it take in a graph as a list of triples of the form (x, y, w) where x and y are vertex labels (perhaps integers) and w is the weight of the edge. There is a subtle but important data processing step to do here: you should make sure that the given graph is connected and that you've given exactly one weight to each edge.*

1.3 The Limitations of Greedy Algorithms

Taking a step back, the settings where greedy algorithms work well are setting where a globally optimal solution can be written in terms of a bunch of locally optimal decisions, and a lot of problems in computer science are hard precisely because the local properties of a problem doesn't give you enough information about the global structure. Greedy algorithms have an 'optimality of substructure' where the greedy solution to some subproblem appears in the optimal solution to the big problem. For example, in the spanning tree problem, you can show that if you take some subtree T' of a minimum spanning tree T, then T' is a minimum spanning tree for its vertex set.

Lots of problems don't have this property. Take, for example, the minimum graph coloring problem. A *proper graph coloring* is an assignment of a color to each vertex such that no edge has two same-colored vertices at each end. The minimum graph coloring problem is to find a proper coloring of a graph using the fewest number of colors. Below is an example of a graph which is four-colorable and a subgraph which contains vertices of all four colors but only requires two colors as its own problem.

Graph coloring is a place where there is a sharp boundary between what the greedy algorithm can promise and what is truly optimal. A straightforward greedy algorithm to color a graph repeatedly arbitrarily chooses a vertex and assigns it a color which is not already used by one of its neighbors. Letting Δ denote the maximum degree of any vertex in the graph, it's straightforward to show that this algorithm never uses more than $\Delta+1$ colors. However, $\Delta+1$ may be far larger than the number of required colors.

Exercise 6. *Prove that the greedy algorithm never uses more than $\Delta+1$ colors for a graph with maximum degree Δ.*

Large bipartite graphs can be two-colored, but the greedy algorithm will use far more than that if the vertices are picked in the wrong order, in which case the solution will not be optimal. Some clever observations about spanning trees gives us a greedy algorithm which uses at most Δ colors in some cases, but moving beyond this requires much more sophisticated techniques from complexity and graph theory, and we don't know of any efficient algorithm for finding an optimal coloring for graphs requiring at least three colors.

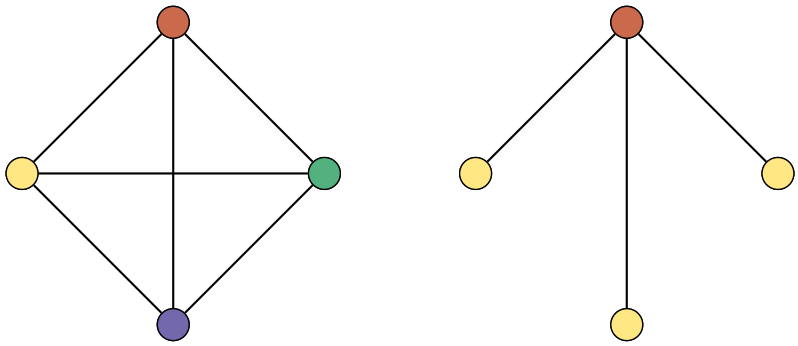


Figure 1. The optimal 4-coloring on the left will not restrict to an optimal 2-coloring on the right

1.4 Further Exercises

Exercise 7. *The refueling problem is as follows: you are driving along a long stretch of highway, starting (for convenience) at mile marker 0 and you are trying to get to a town at mile marker M. Along this highway are several gas stations $1, 2, \ldots, k$ at mile markers $m_1, m_2, \ldots, m_k$. Your car can travel D miles before running out of gas.*

Consider the greedy algorithm where you first check if you are within D miles of M. If so, just drive to M. Otherwise, drive to the furthest station within D miles of your current location and refill your gas tank, then repeat.

1. *Prove that this algorithm gets you to M while refueling the fewest number of times. Hint: a proof similar to that of the correctness of Prim's Algorithm, where you find a contradiction by assuming that there is a strictly smaller number of refuelings that doesn't use the greedy solution is a good way to do it.*
2. *Implement this in Python. Your algorithm should take in a destination M, a maximum driving distance D, and a list of numbers $m_1, m_2, \ldots$ representing the locations of the gas stations. It should output either the list of stations used to reach M or say that it is impossible to do so.*

Exercise 8. *The task scheduling problem takes as input a list of tasks with start and finish times, where we write (s_i, f_i) for the start and finish times of task i. We begin working at time 0 and end at time T and the goal is to find the largest set of tasks that can be accomplished in this time such that the finish time of the last task is at or before T and no two tasks can be worked on concurrently.*

1. *Give a greedy algorithm for this problem.*
2. *Prove the correctness of the algorithm.*
3. *Implement it in Python. Your algorithm should take an end time T and a list of pairs (s_i, f_i). It should output the collection of tasks in the optimal set.*

Exercise 9. *The Traveling Salesperson Problem (TSP) takes as input a weighted graph and the goal is to find a cycle in the graph which visits every vertex such that the total weight of the edges in the cycle is as small as possible. This problem has critical applications in domains like shipping and transportation and is one of the most difficult problems in computer science.*

Here is a greedy algorithm for TSP. We'll assume that the graph given has an edge between every pair of vertices so we can at least guarantee that the algorithm outputs a cycle which visits every vertex.

Algorithm 4 A Greedy Approach to TSP

Input: A graph $G = \langle V, E \rangle$ with edge weights w_{uv}.
Output: A cycle $C = v_1, v_2, \ldots, v_n, v_1$ which visits every vertex G.

```
C ← v_1, i = 1                          ▷ Initialize C at an arbitrary vertex v_1
while C does not visit every vertex do
    Append to C the neighbor of v_i which is not already in C and has the edge of smallest weight
joining it to v_i
    i ← i + 1
return C.
```

1. *Why does this algorithm not find a minimum weight cycle? Give an example of a graph where the algorithm finds a suboptimal cycle.*
2. *Show that the weight of an optimal solution to TSP is at least the weight of a minimum spanning tree of the graph.*
3. *A graph is called **metric** if, given any three vertices x, y, and z, $d(x, y) + d(y, z) \geq d(x, z)$, where $d(u, v)$ is the total weight of the minimum weight path connecting u and v. In other words, a **metric graph** is one where the pairwise shortest paths define a valid metric. Show that on a metric graph, the weight of an optimal solution to TSP is at most twice the weight of a minimum spanning tree of the graph.*