Greedy Algorithms

Topics

1. Introduction
2. Does Greedy always work?
3. Problems

Introduction

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; there are simpler and more efficient algorithms.
A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Greedy is a strategy that works well on optimization problems with the following characteristics:

1. **Greedy-choice property:** A global optimum can be arrived at by selecting a local optimum. The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution.

2. **Optimal substructure:** An optimal solution to the problem contains an optimal solution to subproblems.

Applications of Greedy Algorithms

1. Prim's MST Algorithm
2. Dijkstra's Algorithm
3. Kruskal's MST Algorithm
4. Huffman Coding

Does Greedy always work?

Since the optimal-substructure property is exploited by both greedy and dynamic-programming strategies, one might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices, or one might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

Knapsack Problem

1. **0-1 knapsack problem**
   A thief robbing a store finds N items in the store; the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take? (This is called the 0-1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of

an item or take an item more than once.)

2. **Fractional knapsack problem**
   The setup is the same, but the thief can now take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot, while an item in the fractional knapsack problem is more like gold dust.

Example-1

Knapsack Size = 50	Item-1	Item-2	Item-3
Value	60	100	120
Weight	10	20	30

- Maximum profit for **fractional knapsack** is 240$ [10 units of Item-1, 20 units of Item-2, 20 units of Item-3]
- Maximum profit for **integer knapsack** is 220$ [Item-3 and Item-2]

Solution

1. **Greedy Approach:** Pick elements in non-increasing order of value/unit and fill the knapsack.
   For above example we can veirfy that for both the knapsack problems, we get the optimal solution using Greedy Approach. So far, greedy seems to work for both.

Example-2

Knapsack Size = 30	Item-1	Item-2	Item-3	Item-4
Value	80	90	60	35
Weight	10	15	12	7

For integer knapsack, using greedy approach gives a maximum profit of $170 [Item-1 and Item-2]. But the optimal solution for this example is $175 [Item-1, Item-3 and Item-4]

Conclusion: Greedy doesnt always work. Greedy works for Fractional Knapsack but fails for Integer Knapsack. Integer Knapsack is solved using dynamic programming, which is discussed in the next section.

Problems

1. **Activity-selection problem**
   You are given N activities with start and finish times. Select the max number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.
   Example:
   Consider the following 6 activities.
   start[ ] = {1, 3, 0, 5, 8, 5};
   finish[ ] = {2, 4, 6, 7, 9, 9};

   The maximum set of activities that can be executed by a single person is 4 – Activity0, Activity1, Activity3, Activity4.

   Solution: Sort the activities based on finish time and pick activities in order such that they do not overlap.
   Refer https://en.wikipedia.org/wiki/Activity_selection_problem#Proof_of_optimality for proof on why greedy choice based on finish time works here.

Other References:

1. https://www.topcoder.com/community/data-science/data-science-tutorials/greedy-is-good/
2. http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap17.htm