# **Chapter 4: Greedy Algorithms**

## **Introduction**

**The** **greedy method** is similar to dynamic programming, in that **it is used to solve optimization problems**. Superficially, it seems similar to dynamic programming, in that we expect an optimal solution to consist of optimal solutions to sub-problems. In dynamic programming, though, at each step we make a choice among the various solutions to a sub-problem to pick the optimal one. This might require changing to a more optimal solution.

In the greedy method, **at each step we simply make the most optimal choice we can** based on where we are at. **We never go back and change decisions in the greedy method.** For example, a greedy method to find the shortest path from one vertex to another vertex would simply pick the shortest edge at each step. This will probably not find the shortest path. So, the greedy method does not always work.

## **Coin Changing Problem**

Let's look at a problem that may or may not be solvable via the greedy method.   The problem is that of giving a customer change using the fewest number of coins.   For example, if the change were 26 cents, 1 quarter and 1 penny is better than 26 pennies. We can picture an algorithm that works like this:

**while (change is not equal to desired change)  
{  
    grab the largest coin**

**if (change + coin > desired change)  
        reject the coin  
        do not consider that size of coin again  
    else  
        accept the coin     
}**

If the desired change is 26 cents, we first pick up a quarter.  The next time we try to pick up a quarter, we reject it.  We also reject dimes and nickles.  When we pick up a penny, that gives us the right amount of change and we're done.

## **Greedy Method, Part 2**

Any algorithm that uses the **greedy method goes through three stages:**

1. **Selection procedure**
2. **Feasibility check**
3. **Solution check**

***Selection Procedure***

* This is where we make our choice based on the most optimal choice available to us right now.  We do not consider any possible future choices.

***Feasibility Check***

* This is where we check to see if the solution represented by our last selection is feasible.  A solution is not feasible if it cannot be part of a solution to the larger problem.

***Solution Check***

* This is where we check to see if we have solved the larger problem yet.

## **Coin Changing, Part 2**

Does the greedy method always produce an optimal solution for the coin changing problem?  What if we add in a 12 cent coin?  Does it still produce optimal solutions?  Use it to make 16 cents worth of change.

## **Minimum Spanning Trees**

A common problem in graphing is to determine the subset of a graph that contains all the vertices, but only enough edges to result in a connected graph with no cycles (e.g., a tree).  This is called a spanning tree.  You can use a **depth first search** or a **breadth first search** to construct a spanning tree, by simply adding vertices and edges to the spanning tree as they are crossed, stopping when you have added all the vertices.

Quite often, we care more about the spanning tree with the minimum total edge cost, or the minimum spanning tree.  A minimum spanning tree can be used to help people who are building a network of some kind (**a telecommunications network, a system of plumbing, etc**) to use the least amount of physical cable or pipe.

To calculate all the possible spanning trees and compare them to find the minimum spanning tree would be a factorial, **O(n!), algorithm in the worst case**.  The greedy method is used in different ways to more efficiently construct minimum spanning trees out of graphs.  We will concern ourselves with weighted, undirected graphs. We'll look at two different greedy algorithms for constructing a minimum spanning tree:

1. **Prim's algorithm**
2. **Kruskal's algorithm**

## **Prim's Algorithm**

Prim's algorithm constructs a minimum cost spanning tree.  Rather than considering any edges, prim's algorithm makes certain that at every stage the set of edges picked constructs a tree.  We start with any vertex as the root of the tree.

**The selection procedure** is then to select the minimum edge that is connected to a vertex in the tree, giving preference to edges connected to lower numbers vertices in the tree if there is a tie.

**The feasibility check** is to make certain that the edge does not lead to a vertex already in the tree (**this prevents cycles**).

**The solution check** is to see whether we have all the vertices in the tree.  If so, we have constructed a minimum spanning tree.

The following data structures are used:

* F is a set of edges, initially empty
* V is a set of vertices containing all the vertices in the graph
* Y is a set of vertices, initially containing only one vertex from the graph

while (problem is not solved)  
{  
**select a vertex from V - Y** (this ensures that cycles are not created) such that the edge from Y to that vertex is minimum

**add vertex to Y  
    add edge to F**

    if (Y == V)  
        problem is solved  
}

The worst case order of complexity is **O(n2).** This would seem to indicate that Prim's algorithm is better than Kruskal's algorithm because Kruskal’s worst case is given by **O(n2 lg n)**. However, that is only true for the worst case. For graphs with fewer edges, Kruskal's algorithm performs better.

## **Kruskal's Algorithm**

Kruskal's algorithm concerns itself with choosing which edges will be in the minimum spanning tree.  It does this by **making a greedy choice and selecting the minimum edge in the graph**.  That edge and the vertices connected to it are added to the minimum spanning tree.  It then picks the next minimum edge, and so on.  **That's the selection procedure.**

**The feasibility check comes in** because we **cannot have cycles** in the minimum spanning tree.  So, if selecting an edge would result in a cycle, then **that edge is not part of a valid solution**, so we reject it.

**The solution check** is simply to recognize **when we have connected all the vertices**.   At that point, we have a minimum spanning tree.

In this algorithm, the following data structures are used:

* F is a set of edges that is initially empty
* E is a set of edges that initially contains all the edges

We also have n sets, each containing exactly one vertex from the graph. We first sort the set E in order by cost, so that we can choose the minimum cost edge from it (a heap is often used for this, so that we can remove the minimum cost edge every time)

while (the problem is not solved)  
{  
    select next edge from E (this removes the edge from E)

    if (the edge connects vertices in two disjoint sets)  
    {  
        merge the sets containing the two vertices  
        add the edge to F  
    }

    if (all vertices are in one set)  
        the problem is solved  
}

The worst case order of complexity for this algorithm is **O(n2 lg n)**.   For graphs with a relatively **few number of edges**, the order of complexity is **O(n lg n).**

## **Dijkstra's Algorithm**

Recall that we talked about **Floyd's all-pairs shortest path algorithm, a dynamic programming approach to finding the** **shortest path from every vertex to every other vertex in a graph**.  Floyd's algorithm was **O(n3).**

Now we'll talk about **Dijkstra's algorithm for finding the shortest path from one particular vertex to every other vertex**.  This is called a single-source shortest path algorithm, and is a greedy algorithm.

Here are the basic steps. We use three sets, Y = {v1}, F = {}, V = {all vertices}

Choose the shortest path from a vertex in Y to a vertex in V - Y.  The path can only use vertices in Y as intermediate nodes (i.e., every vertex in the path must be in Y except for the ending vertex, which must be in V).  Add the ending vertex to Y, and add the edge chosen to F.

Note that we are looking for shortest paths, not shortest edges.  **It may well be that the new edge we add is not the shortest edge remaining, because the shortest edge would result in a longer path.**Repeat, stopping only when Y = = V.

Pages 170-172 in your book has the full algorithm.