# Chapter 4

# The Greedy Approach

## Preview

- Overview
- Lecture Notes
  - Minimum Spanning Tree
    - Prim's Algorithm
    - Kruskal's Algorithm
    - Comparing Prim's Algorithm with Kruskal's Algorithm
    - Final Discussion
- Quick Check
  - Dijkstra's Algorithm for Single-Source Shortest Path
  - Scheduling
    - Minimizing Total Time in the System
    - Scheduling with Deadlines
  - Huffman Code
    - Prefix Codes
    - Huffman's Algorithm
- Quick Check
  - The Greedy Approach versus Dynamic Programming: The Knapsack Problem
    - A Greedy Approach to the 0-1 Knapsack Problem
    - A Greedy Approach to the Fractional Knapsack Problem
    - A Dynamic Programming Approach to the 0-1 Knapsack Problem
    - A Refinement of the Dynamic Programming Algorithm for the 0-1 Knapsack Problem
- Quick Check

- Classroom Discussion
- Homework
- Keywords
- Important Links

# Overview

The Charles Dickens character Ebenezer Scrooge is a fine illustration of the greedy approach. Each day Scrooge grabbed as much gold as he could. Scrooge only changed his ways after the Ghosts of Christmas Past and Future informed him of the consequences.

The **greedy algorithm** grabs data items in sequence, each time taking the item that is deemed to be "best," without regard to the choices it has made before or to the choices that it might make in the future. Despite the negative impression, they often lead to simple and efficient solutions.

A greedy algorithm arrives at a solution by making a sequence of choices, each of which simply looks best at the moment. The Change problem discussed in the text shows that a greedy algorithm does not guarantee an optimal solution.

A greedy algorithm starts with an empty set and adds items to the set in sequence until the set represents a solution to an instance of a problem. Each iteration in the greedy algorithm consists of the following components:

- A **selection procedure**, chooses the next item to add to the set. The selection is preformed according to a greedy criterion that satisfies some locally optimal consideration at the time.
- A **feasibility check**, determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.
- A **solution check**, determines whether the new set constitutes a solution to the instance.

# Lecture Notes

## Minimum Spanning Tree

Before discussing minimum spanning trees, let's continue our review of graph theory. A graph is **undirected** when its edges do not have direction. This is represented pictorially by an absence of arrows in the edges. Because the edges do not have direction, we say an edge is between two vertices. A **path** in an undirected graph is a sequence of vertices such that there is an edge between each vertex and its successor. An undirected graph is called **connected** if there is a path between every pair of vertices. All the graphs in Figure 4.3 are connected. If we remove an edge as in Figure 4.3, the graph is no longer connected.

In an undirected graph, a path from the vertex to itself, which contains at least three vertices and in which all intermediate vertices are distinct, is called a **simple cycle**. An undirected graph with no simple cycles is called **acyclic**. A **tree** is an acyclic, connected, undirected graph. A **rooted tree** is defined as a tree with one vertex designated as the root.

Consider the problem of removing edges from a connected, weighted, undirected graph *G* to form a sub-graph such that all the vertices remain connected, and the sum of the weights on the remaining edges is as small as possible. This kind of problem has numerous applications. A **spanning tree** for *G* is a connected subgraph that contains all the vertices in *G* and is a tree. A connected subgraph of minimum weight must be a spanning tree, but not every spanning tree has minimum weight. A spanning tree of minimum weight is called a **minimum spanning tree**. A graph can have more than one minimum spanning tree. Prim's and Kruskal's algorithms always produce minimum spanning trees.

## Prim's Algorithm

Prim's algorithm starts with an empty subset of edges *F* and a subset of vertices *Y* initialized to contain an arbitrary vertex. We will initialize *Y* to {v1}. A vertex nearest to *Y* is a vertex in *V – Y* that is connected to a vertex in *Y* by an edge of minimum weight. The vertex that is nearest to *Y* is added to *Y* and the edge is added to *F*.

The selection procedure and feasibility check are done together because taking the new vertex from *V – Y* guarantees that a cycle is not created.

This high-level algorithm works fine for a human creating a minimum spanning tree for a small graph from a picture of the graph. For the purposes of writing an algorithm that can be implemented in a computer language, we need to describe a step-by-step procedure. This procedure is described in detail in the text.

Prim's algorithm produces a spanning tree. But is it minimal? Although greedy algorithms are easier to develop than dynamic programming algorithms, it is usually more difficult to determine whether or not the greedy algorithm produces an optimal solution. For a dynamic algorithm, we need only show that the principle of optimality applies. We usually need a formal proof for a greedy algorithm. Please refer to the text for the demonstration of the proof.

## Kruskal's Algorithm

Kruskal's algorithm for the Minimum Spanning Tree problem starts by creating disjoint subsets of *V*, one for each vertex and containing only that vertex. It then inspects the edges according to non-decreasing weight. If an edge connects two vertices in disjoint subsets, the edge is added, and the subsets are merged into one set. This process is repeated until all the subsets are merged into one set.

To write a formal version of Kruskal's algorithm, we need a disjoint set abstract data type. Such a data type is implemented in Appendix C. This consists of data types *index* and *set pointer*, and routines *initial, find*, *merge,* and *equal,* such that if we declare

```
index i ;
set pointer p, q ;
```

Then

- initial(n) initializes n disjoint subsets, each of which contains exactly one of the indices between 1 and *n*.
- *p = find (i)* makes *p* point to the set containing index *i*.
- *merge (p, q)* merges the two sets, to which *p* and *q* point, into the set.
- *equal (p, q*) returns true if *p* and *q* both point to the same set.

## Comparing Prim's Algorithm with Kruskal's Algorithm

For a graph whose number of edges m is near the low end of these limits, Kruskal's algorithm is faster. Prim's algorithm is faster for a graph whose number of edges is near the high end (a highly connected graph).

## Final Discussion

The time complexity of an algorithm sometimes depends on the data structure used to implement it. For Prim's algorithm, we have many implementations. First, using heaps, we have an $\Upsilon$(mlogn) implementation. Second, using the Fibonacci heap, we have the fastest implementation of Prim's algorithm which is $\Upsilon(m+n\log n)$. The history of the Minimum Spanning Tree problem is discussed in Graham and Hell (1985). Other algorithms for the problem can be found in Yao (1975) and Tarjan (1983).

# Quick Check

1. Every spanning tree has minimum weight. (True or False )
   Answer: false
2. A vertex ___ to Y is a vertex in V – Y that is connected to a vertex in Y by an edge of minimum weight.
   Answer: nearest
3. Prim's Algorithm always produces a minimum spanning tree. (True or False )
   Answer: true
4. Index and set-pointer are abstract ____ used by ____ algorithm.
   Answer: data types, Kruskal's

# Dijkstra's Algorithm for Single-Source Shortest Path

In Section 3.2, we developed an algorithm for determining the shortest paths from each vertex to all other vertices in a weighted, directed path. If we want to know only the shortest paths from one particular vertex to all the others, that algorithm would be overkill. We can use a greedy algorithm, called the Single-Source Shortest Paths problem, which was developed by Dijkstra (1959). Our presentation of the algorithm assumes that there is a path from the vertex of interest to each of the other vertices. There is a simple modification to the algorithm that works when this is not the case.

We initialize a set Y to contain only the vertex whose shortest paths are to be determined. We initialize a set F of edges to being empty. First we choose a vertex v that is nearest to *v1*, add it to *Y*, and add the edge <*v1, v*> to *F*. That edge is clearly a shortest path from *v1* to *v*. Next we check the paths from *v1* to the vertices in *V – Y* that allow only vertices in *Y* as intermediate vertices. A shortest of these paths is a shortest path. The vertex at the end of such a path is added to *Y*, and the edge that touches that vertex is added to *F*. This procedure is continued until *Y* equals *V*, the set of all vertices.

As was the case for Prim's algorithm, this high-level algorithm works only for a human solving an instance by inspection on a small graph. The more detailed algorithm in the text is represented by a weighted graph with a two-dimensional array as shown in Section 3.2. This algorithm is very similar to Algorithm 4.1 (Prim's). The difference is that instead or the arrays *nearest* and *distance*, we have the arrays *touch* and *length*. The algorithm is depicted in the text (4.3). This algorithm determines only the edges of the shortest paths. It does not produce the length of those paths. These lengths can be obtained from the edges.

As was the case for Prim's algorithm, Dijkstra's algorithm can be implemented using a heap of a Fibonacci heap.

# Scheduling

Imagine that a hair stylist has several customers waiting for different treatments. Not all of the treatments take the same amount of time, but the stylist knows how long each treatment takes. The stylist's goal is to schedule the clients so that the customers have a minimum amount of waiting time and he or she spends a minimum amount of time servicing the customers. The total amount of waiting time and service time is called **time in the system**. This type of problem has many application areas.

Another scheduling problem that occurs happens when each job takes the same amount of time to complete, but each job also has a deadline by which it must start to yield a profit associated with the job. This type of problem is called **scheduling with deadlines**.

## Minimizing Total Time in the System

Simple solution: Consider all possible schedules and compute the minimum total time in the system.

Example: Assume there are 3 jobs and the service times for them are t1=5, t2=10, t3=4
Assume the schedule is [1, 2, 3], then the total time in the system is:

$$\underbrace{5}_{\text{Time for job1}} + \underbrace{(5+10)}_{\text{Time for job2}} + \underbrace{(5+10+4)}_{\text{Time for job3}}$$

| Schedule | Total time in the system |
|---|---|
| [1, 2, 3] | 39 |
| [1, 3, 2] | 33 |
| [2, 1, 3] | 44 |
| [2, 3, 1] | 43 |
| [3, 1, 2] | 32 |
| [3, 2, 1] | 37 |

[3, 1, 2] of total time in system 32 is optimal.
If we have *n* jobs, then the number of different schedules is *n*!

The algorithm that considers all possible schedules is factorial-time. Intuitively, it seems that this schedule would be optimal because it gets the shortest jobs done first. See the text for a high-level greedy approach to this problem. So we notice that the only schedule that minimizes the total time in the system is one that schedules jobs in non-decreasing order by service time. See the text for proof of this theorem.

## Scheduling with Deadlines

In this scheduling approach:

- Each job takes one unit of time
- If job starts before or at its deadline, profit is obtained, otherwise no profit
- Goal is to schedule jobs to maximize the total profit
- Not all the jobs have to be scheduled.

Consider this example:

| Job | Deadline | Profit |
|---|---|---|
| 1 | 2 | 30 |
| 2 | 1 | 35 |
| 3 | 2 | 25 |
| 4 | 1 | 40 |

Job 1 has deadline 2, therefore it can start at time 1 or 2.
Job 2 has deadline 1, therefore it can start at time 1.
Job 3 has deadline 2, therefore it can start at time 1 or 2.
Job 4 has deadline 1, therefore it can start at time 1.

| Schedule | Total Profit |
|---|---|
| [1, 3] | 30 + 25 = 55 |
| [2, 1] | 35 + 30 = 65 |
| [2, 3] | 35 + 25 = 60 |
| [3, 1] | 25 + 30 = 55 |
| [4, 1] | 40 + 30 = 70 |
| [4, 1] | 40 + 25 = 65 |

Definitions:

- A sequence is a **feasible sequence,** if all the jobs in the sequence start by their deadlines. e.g., [4, 1] is a feasible sequence, where as [1, 4] is not.
- A set of jobs is called **feasible set** if there is at least one feasible sequence for the jobs in the set. E.g., {1, 4} is a feasible set, where as {2, 4} is not.
- A feasible sequence with maximum total profit is called **optimal sequence.**
- The set of jobs in the optimal sequence is called **optimal set of jobs**.

Example 4.3 considers all the schedules, and it takes factorial time. As you can see, only the job with the greatest profit has been scheduled. This suggests that a reasonably greedy approach to solving the problem would be to first sort the jobs in nonincreasing order by profit and then inspect each job in sequence and add it to the schedule if possible. See the text for more examples.

## Huffman Code

Even though the capacity of secondary storage devices keeps getting larger, and their cost keeps getting smaller, the devices continue to fill up due to increased storage demands. Given a data file, it would therefore be desirable to find a way to store the file as efficiently as possible. The problem of **data compression** is to find an efficient method for encoding a data file. Next, we discuss the encoding method, called **Huffman code,** and a greedy algorithm for encoding a given file.

A common way to represent a file is to use a **binary code**. In such a code, each character is represented by a unique binary string, called the **codeword.** A **fixed-length** binary code represents each character using the same number of bits. If our character set is {*a,b,c*}, we could use two bits to code each character, as follows:

a: 00 b: 01 c: 11

Given this code, if our file is

ababcbbbc,

our encoding is

000100001110101011111.

We can obtain a more efficient encoding using **variable-length binary code**. Such a code can represent different characters using different numbers of bits.

The code

a: 10 b: 0 c: 11

Would be encoded as

1001001100011.

With this encoding, it takes only 13 bits to represent the file, where it took 18 bits to represent the previous file.

Given a file, the Optimal Binary Code problem is to find a binary code for the characters in the file, which represents the file in the least number of bits. First, we will discuss prefix codes, and then we will develop Huffman's algorithm for solving this problem.

## Prefix Codes

One particular type of variable-length code is a **prefix code**. In a prefix code, no codeword for one character constitutes the beginning of the codeword for another character. A fixed-length code is also a prefix code. Every prefix code can be represented by a binary tree whose leaves are the characters that are to be encoded. The advantage of a prefix code is that we need not look ahead when parsing the file. To parse, we start at the first bit on the left in the file and the root of the tree. We sequence through the bits, and go left or right down the tree depending on whether a 0 or 1 is encountered. When we reach a leaf, we obtain the character at that leaf; then we return to the root and repeat the procedure starting with the next bit in sequence.

## Huffman's Algorithm

Huffman developed a greedy algorithm that produces an optimal binary character code by constructing a binary tree corresponding to an optimal code. A code produced by this algorithm is called a **Huffman code**.

We represent a high-level version of the algorithm. Because it involves constructing a tree, we need to be more detailed in our other high-level algorithms. We have the following type declaration:

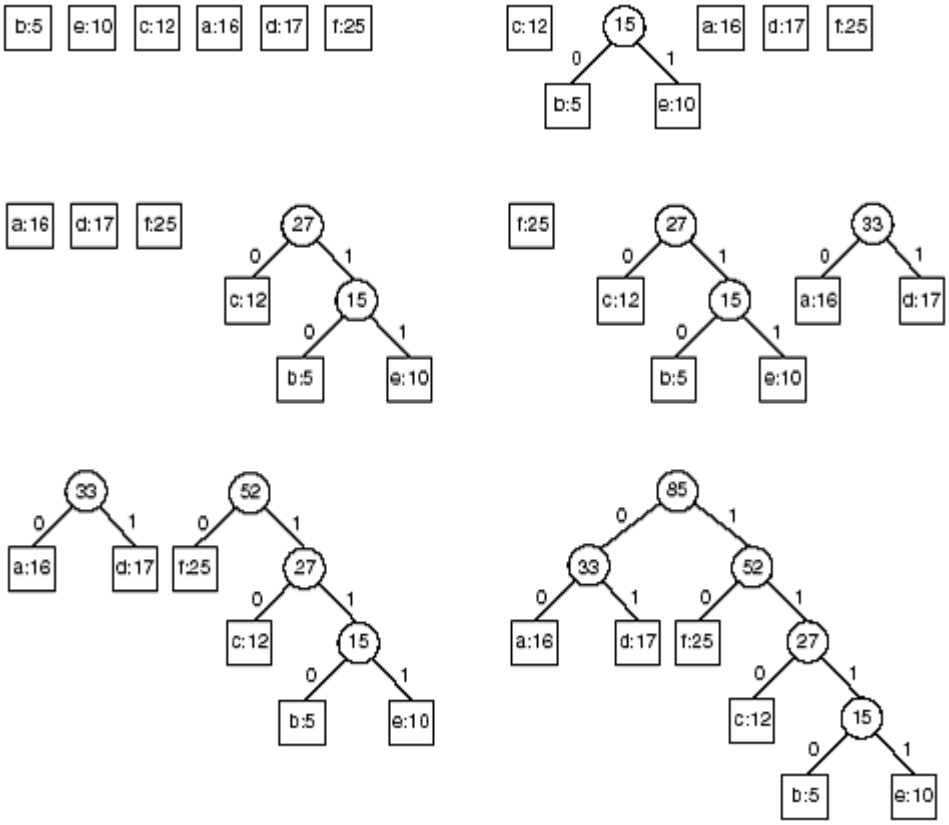
[See declaration on page 186]

We also need to use a **priority queue**. In a priority queue, the element with the highest priority is always removed next. In this case, the element with the highest priority is the character with the lowest frequency in the file. The priority queue can be implemented as a linked list, but more efficiently as a heap.

Example:

| Character | Frequency |
|---|---|
| A | 15 |
| B | 5 |
| C | 12 |
| D | 17 |
| E | 10 |
| F | 25 |

We will construct the tree (using Huffman's Algorithm)




## Quick Check

1. The time spent both waiting and being served is called the ____.
   Answer: time in the system
2. The only schedule that minimizes the total time in the system is one that schedules jobs in non-increasing order by service time. (True or False)
   Answer: False
3. A sequence whose jobs start by their deadlines is called ____.
   Answer: feasible
4. A unique binary string called the ____.
   Answer: codeword

# The Greedy Approach versus Dynamic Programming: The Knapsack Problem

The greedy approach and dynamic programming are two ways to solve optimization problems. A problem can be solved using either approach. The dynamic programming algorithm is overkill in that it produces the shortest paths from all sources. There is no way to modify the algorithm to produce more efficiently only the shortest paths from a single source. But when the greedy approach solves a problem, the result is a simpler, more efficient algorithm. On the other hand, it is usually more difficult to determine whether a greedy algorithm always produces an optimal solution. A proof is needed to show that a particularly greedy algorithm always produces an optimal solution, whereas a counterexample is needed to show that it is not.

## A Greedy Approach to the 0-1 Knapsack Problem

In this example, a thief carrying a knapsack breaks into a jewelry store. The knapsack will break if the total weight exceeds the maximum weight, *W*. Each item in the knapsack has value and weight. The thief's dilemma is to maximize the total number of items while not making the total weight exceed *W*.

The obviously greedy strategy is to steal the items with the largest profit first, steal them in nonincreasing order according to profit. This strategy does not work if the most valuable item has a large weight in comparison to its profit.

Another greedy strategy is to steal the lightest items first. This strategy fails if the lightest items have smaller profits.

The most efficient strategy is for the thief to steal the items that have the largest profit per unit weight first. So, we order the items in nonincreasing order according to profit per unit weight and select them in sequence. An item is put in the knapsack if it does not bring the total weight above *W*.

## A Greedy Approach to the Fractional Knapsack Problem

In the Fractional Knapsack problem, the thief does not have to steal all of an item, but rather can take any fraction of the item.

The greedy approach to the fractional knapsack problem yields the optimal solution:

$50 + $140 + (5/10) *60 = $220 (The thief takes 5/10 of item 2)

5 is the remaining capacity of the knapsack

The greedy algorithm never wastes any capacity in the Fractional Knapsack problem as it does in the 0–1 Knapsack problem. It always leads to an optimal solution.

## A Dynamic Programming Approach to the 0 –1 Knapsack Problem

If we can show that the principle of optimality applies, we can use dynamic programming to solve the 0 –1 Knapsack problem.

Let *A* be an optimal subset of *n* items. There are two cases:

- If *A* contains item *n*, then the total profit of items in *A* is equal to $p_n$ plus the optimal profit obtained from the first *n* –1 items, where the total weight cannot exceed $W - w_n$.
- If *A* does not contain item n, then the total profit of items in *A* is equal to the optimal subset of the first *n –1* items.

### A Refinement of the Dynamic Programming Algorithm for the 0-1 Knapsack Problem

The fact that the previous expression for the number of array entries computed is linear in *n* can mislead one into thinking that the algorithm is efficient for all instances containing *n* items. That is not the case. There is no relationship between *n* and *W*. When W is extremely large in comparison to n, this algorithm is worse than the brute-force algorithm that simply considers all subsets. See the text for more details.

As is the case for the Traveling Salesperson problem, no one has ever found an algorithm for the 0–1 knapsack problem whose worst-case time complexity is better than exponential, yet no one has proven that such an algorithm is not possible.

## Quick Check

1. It is very easy to determine whether a greedy algorithm always produces an optimal solution (True or False)
   Answer: false
2. In the Knapsack Problem, an obvious solution is to steal the lightest items fist, this fails when the light items have ____ profits compared with their weights.
   Answer: small
3. In the fractional knapsack problem the thief, can take any ____ of the item.
   Answer: fraction

## Classroom Discussion

- Prim's Algorithm always produces a minimum spanning tree. Discuss
- A binary tree corresponding to an optimal binary prefix code must be full. Discuss
- Can Dijkstra's algorithm be used in a graph with negative weights? Justify.

## Homework

Assign Exercises 2, 17, 25, and 28.

## Keywords

- **Acyclic –** a graph that has no cycles.
- **Binary code –** a common way to represent a file.
- **Branch –** a branch with root v in tree T is the sub-tree whose root is v.
- **Codeword –** a unique binary string.
- **Connected Graph –** if there is a path between every pair of vertices.
- **Data compression –** efficient method for encoding a file.

- **Factorial time –** the time it takes to raise *n* to the *n*th power.
- **Feasibility check –** determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.
- **Feasible sequence –** if all the jobs in the sequence start by their deadlines.
- **Feasible set –** a set of jobs such that there exists at least one feasible sequence for the jobs in the set.
- **Fixed-length binary code –** represents each character using the same number of bits.
- **Greedy algorithm –** an algorithm that arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment.
- **Huffman code –** an encoding method.
- **Minimum spanning tree –** a spanning tree of minimum weight.
- **Optimal sequence –** a feasible sequence with maximum total profit.
- **Optimal set of jobs –** set of jobs in an optimal sequence.
- **Path** – sequence of vertices such that there is an edge/arc from each vertex to its successor.
- **Prefix code –** a particular type of variable-length code.
- **Priority Queue –** a queue such that the element with the highest priority is removed next.
- **Promising –** a set of edges *E* is said to be *promising* if edges can be added to it so as to form a minimum spanning tree.
- **Rooted tree –** a tree with one vertex designated as the root.
- **Scheduling with deadlines –** when each job takes the same amount of time to complete but has a deadline by which it must start to yield a profit associated with the job.
- **Selection procedure –** chooses next item to add to set.
- **Siblings –** two nodes are called siblings if they have the same parent.
- **Simple Cycle –** a path from a vertex to itself, which contains at least three vertices and in which all intermediate vertices are distinct.
- **Solution check –** determines whether the new set constitutes a solution to the instance.
- **Spanning tree –** for a graph G is a connected sub-graph that contains all the vertices in *G* and is a tree.
- **Time in the system –** time spent both waiting and being served.
- **Tree –** an acyclic, connected, undirected graph.
- **Undirected graph –** graph with edges who do not have direction.
- **Variable-length binary code –** represent different characters using different number of bits.

## Important Links

- http://lcm.csa.iisc.ernet.in/dsa/node162.html (Single Shortest Paths Problem)
- www.nist.gov/dads/ (Dictionary of Algorithms and Data Structures)
- www.cs.pitt.edu/~kirk/algorithmcourses/ (Algorithms courses on the Internet)