Section 3: Union
Find/Greedy Algorithms

Kat Zhang
February 2021
Table of Contents

- Union-Find
  - What is it and why do we need it?
  - What operations does it handle?
  - How do we optimize it?

- Greedy Algorithms
  - What does it mean for an algorithm to be greedy?
  - Recognizing greedy vs. not greedy
Union Find
What is Union Find?

- Union Find is a **data structure**, not an algorithm
- What is it used for?
  - Representing and manipulating **disjoint sets**
  - Specifically used in **Kruskal’s algorithm** when merging, finding, etc. sets for particular elements
What operations does Union Find have?

- **MAKESET** → create a set holding x (x is the root of the new tree)
- **UNION(x, y)** → combine/union the two sets containing x and y (basically combining the trees so that they have a single root)
- **FIND(x)** → find the set that x belongs to (find the root of the tree x is in)
- **LINK(x, y)** → (where x, y are roots) → changes the root of one to the other
  - **UNION(x, y) = LINK(FIND(x), FIND(y))**
  - This is defined for our own convenience
What operations does Union Find have?
How do we optimize Union-Find?

- What does it mean to optimize?
  - The deeper the tree, the more time it takes to FIND
  - We want trees to generally be less deep/to minimize the number of operations for the functions of Union Find
Method 1: Union by Rank

When performing a UNION operation, we prefer to merge the shallower tree into the deeper tree using “ranks”

- Rank of x is set to 0 by **MAKESET(x)**
- **LINK** updates rank
  - *If rank(x) = rank(y) = r*: x’s parent points to y and rank(y) = r + 1
  - *Else*: parent of element with smaller rank is updated to point to the parent of element with larger rank
- **Idea**: make trees short when you have to **UNION** them
Method 1: Union by Rank

Union(x, y) with Union by Rank

Link (Find(x), Find(y))

r = 2

x

r = 1

y
Method 1: Union by Rank
Method 2: Path Compression

After performing a **FIND** operation, we can simply attach all the nodes touched directly onto the **root** of the tree.

Idea: if you ever try to **FIND** this element again, it takes **much less time**
Method 2: Path Compression

FIND(x) with Path Compression
Method 2: Path Compression
Greedy Algorithms
What are Greedy Algorithms?

**Definition:** An algorithm that attempts to find the global optimum through making *locally optimal* choices

**Key Point:** Need to be able to show that taking the local optimum keeps you on track to finding the global optimum
Examples

- **Dijkstra’s**: greedily chooses the closest edge to an unexplored vertex
- **Kruskal’s/Prim’s**: greedily choosing the lightest edge
- **Horn Formula**: start with all variables at false, only set true when you need to
- **Huffman Encoding**: using the two least used characters and merging them
- **Set Cover**: greedily choosing the set that covers the largest number of remaining uncovered elements
Horn Formula

**Definition:** A special case of the SAT problem where each clause has at most one positive literal

- \((x \lor y \lor z)\) is not valid in a Horn Formula
- \((x \lor \bar{y})\) is valid in a Horn Formula
- \((y)/(\bar{y})\) are valid in a Horn Formula
Horn Formula

Setup for Algorithm
- Split into pure negative clauses and implications
  - \((x \lor !y \lor !z \lor !w)\) is equivalent to \((y \land z \land w) \rightarrow x\).

Algorithm
- Start with all variables as False
- While there is an implication that is not satisfied, set the necessary variable as true
- Then, look at the pure negative clauses
- If they are satisfied, the Horn Formula is satisfiable
Horn Formula

\[(\overline{x} v \overline{z} v y) \land (\overline{x} v y) \land (x) \land (\overline{x} v \overline{y} v \overline{z})\]

- \(x \land z \rightarrow y\)
- \(x \rightarrow y\)
- \(x \rightarrow x\)

False False True

\(y = \text{True}\) \(\rightarrow\) \(y = \text{True}\)
\(x = \text{False}\) \(\rightarrow\) \(x = \text{True}\)
\(z = \text{False}\) \(\rightarrow\) \(z = \text{False}\)
Huffman Encoding

**Definition:** Finding a bitwise encoding for several characters such that no character’s encoding is the prefix of another’s that minimizes the number of bits used

- Do this by taking into account the **frequencies** at which the characters occur

**Constructing the Encoding:** represent the encoding as a binary tree where the leaves are the encodings of characters
Huffman Encoding

Algorithm

- Bottom-up construction of a tree that gives the most efficient encoding for the given characters
  - Start with a list of all the symbols: $A, B, C, D, E$ → $DE$ → $F(D)$
  - Take the two least frequently used symbols out of the list: $A, B, C, D, E$
  - Combine these two symbols into a metacharacter with a frequency equal to the sum of the two characters’ frequencies, which you add back to the list
  - Make the metacharacter the parent of the two original characters
  - Repeat until there are no more symbols to be put in the tree
Huffman Encoding

A: 36
B: 43
C: 23
AC: 59
ACB: 102
Set Cover

**Definition:** Given a set of elements $X$ and a set of subsets $S$ whose union is $X$, find the smallest collection of subsets $T$ that lie in $S$ that still covers $X$ (set cover)

**Greedy Algorithm (NOTE: this is an approximation!):** repeatedly include the set that contains the maximum number of yet-uncovered elements

- If $k$ is the size of the smallest set cover, greedy finds a set of size at most $k \cdot \ln(n)$ where $n$ is the number of elements in $X$
Greedy vs Not Greedy: The Knapsack Problem

0-1 Knapsack Problem:
- \( N \) items
- Item \( i \) is worth \( v_i \) and has weight \( w_i \)
- Can carry at most \( W \) pounds
- Must either take an item or leave it
- How do you maximize the value?

Fractional Knapsack Problem:
- \( N \) items
- Item \( i \) is worth \( v_i \) and has weight \( w_i \)
- Can carry at most \( W \) pounds
- Can take fractional amounts of items (i.e. \( \frac{1}{2} \))
- How do you maximize the value?
Greedy or Not Greedy: Fractional Knapsack Problem

Greedy Strategy for Fractional Knapsack

- Compute the value per pound $\frac{v_i}{w_i}$ for each item $i$
- Take as much as possible of the item with the greatest value per pound
- If you can’t hold more, stop
- If you can, move to the item with the next greatest value per pound
- Repeat until $W$ pounds of items are collected
Why does greedy work here?

Proof:

- Suppose that there exists an optimal solution in you didn’t take as much of some item \( j \) as possible.
- If the knapsack is not full, add some more of item \( j \), and you have a higher value solution \( \rightarrow \text{Contradiction} \).
- We thus assume the knapsack is full.
- There must exist some item \( k \neq j \) with \( \frac{v_k}{w_k} < \frac{v_j}{w_j} \) that is in the knapsack.
- We also must have that not all of \( j \) is in the knapsack.
- We can therefore take a piece of \( k \), with weight \( x \), out of the knapsack, and put a piece of \( j \) with weight \( x \) in.
- This increases the knapsacks value by \( x(\frac{v_j}{w_j} - \frac{v_k}{w_k}) > 0 \) \( \rightarrow \text{Contradiction} \).
Greedy or Not Greedy: 0-1 Knapsack Problem

\[ W = 50 \]
Greedy or Not Greedy: 0-1 Knapsack Problem
Problem 1

Hex is a two player abstract strategy board game in which players attempt to connect opposite sides of a hexagonal board. Hex was invented by mathematician and poet Piet Hein in 1942 and independently by John Nash in 1948. (Wikipedia)

Design an algorithm that checks if the winner exists after every move.
Problem 2

Suppose we want to make change for $n$ cents, using the least number of coins of denominations 1, 10, and 25 cents. Consider the following greedy strategy: suppose the amount left to change is $m$; take the largest coin that is no more than $m$; subtract this coin's value from $m$, and repeat. Either give a counterexample, to prove that this algorithm can output a non-optimal solution, or prove that this algorithm always outputs an optimal solution.
Problem 3

Suppose that we have a set $S = \{a_1, \ldots, a_n\}$ of proposed activities. Each activity $a_i$ has a start time $s_i$ and a finish time $f_i$. We can only run one activity at a time. Your job is to find a maximal set of compatible activities. Which of the following greedy algorithms is correct?

(a) Sort all the activities by their duration and greedily picking the shortest activity that does not conflict with any of the already chosen activities.

(b) Pick the activity that conflicts with the fewer number of remaining activities. Remove the activities that the chosen activity conflicts with. Break ties arbitrarily.

(c) Sort all the activities by their end time and greedily pick the activity with the earliest end time that does not conflict with any of the already chosen activities.