Greedy Algorithms

R. Sekar

Overview

- One of the strategies used to solve *optimization problems*
  - Multiple solutions exist; pick one of low (or least) cost
- *Greedy strategy:* make a locally optimal choice, or simply, what appears best at the moment
- Often, *locally optimality ≠ global optimality*
- So, use with a great deal of care
  - *Always need to prove optimality*
- If it is unpredictable, why use it?
  - *It simplifies the task!*

Making change

Given coins of denominations 25¢, 10¢, 5¢ and 1¢, make change for \( x \) cents \((0 < x < 100)\) using *minimum number of coins.*

**Greedy solution**

\[
\text{makeChange}(x) \\
\text{if } (x = 0) \text{ return} \\
\text{Let } y \text{ be the largest denomination that satisfies } y \leq x \\
\text{Issue } \lfloor x/y \rfloor \text{ coins of denomination } y \\
\text{makeChange}(x \mod y)
\]

- Show that it is optimal
- Is it optimal for arbitrary denominations?

When does a Greedy algorithm work?

**Greedy choice property**

The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution

What does this mean for the coin change problem?

**Optimal substructure**

The optimal solution contains optimal solutions to subproblems.

Implies that a greedy algorithm can invoke itself recursively after making a greedy choice.
**Knapsack Problem**

- A sack that can hold a maximum of $x$ lbs
- You have a choice of items you can pack in the sack
- Maximize the combined “value” of items in the sack

<table>
<thead>
<tr>
<th>Item</th>
<th>Calories/lb</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>1100</td>
<td>5</td>
</tr>
<tr>
<td>Butter</td>
<td>3300</td>
<td>1</td>
</tr>
<tr>
<td>Tomato</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>Cucumber</td>
<td>55</td>
<td>2</td>
</tr>
</tbody>
</table>

0-1 knapsack: Take all of one item or none at all

Fractional knapsack: Fractional quantities acceptable

**Greedy choice:** pick item that maximizes calories/lb

Will a greedy algorithm work, with $x = 5$?

---

**Fractional Knapsack**

**Greedy choice property**

Proof by contradiction: Start with the assumption that there is an optimal solution that does not include the greedy choice, and show a contradiction.

**Optimal substructure**

After taking as much of the item with $j$th maximal value/weight, suppose that the knapsack can hold $y$ more lbs. Then the optimal solution for the problem includes the optimal choice of how to fill a knapsack of size $y$ with the remaining items.

Does not work for 0-1 knapsack because greedy choice property does not hold.

0-1 knapsack is NP-hard, but a pseudo-polynomial algorithm is available.

---

**Spanning Tree**

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices $V$ in the graph
- A subset of $E$ such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by

- A maximal subset of $E$ such that the subgraph has no cycles
- A subset of $E$ with $|V| - 1$ edges such that the subgraph is connected
- A subset of $E$ such that there is a unique path between any two vertices in the subgraph

---

**Minimal Spanning Tree (MST)**

A spanning tree with *minimal cost*. Formally:

**Input:** An undirected graph $G = (V, E)$, a cost function $w : E \to \mathbb{R}$.

**Output:** A tree $T = (V, E')$ such that $E' \subseteq E$ that minimizes $\sum_{e \in E'} w(e)$
### Minimal Spanning Tree (MST)

![Graph with nodes A, B, C, D, E, F and edges connecting them with weights 1, 2, 3, 4, 5, 6.](image)

### Kruskal’s algorithm

- Start with the empty set of edges
- Repeat: add lightest edge that doesn’t create a cycle

Adds edges $B-C, C-D, C-F, A-D, E-F$

![Graph showing the minimum spanning tree found by Kruskal’s algorithm.](image)

### Kruskal’s: Correctness (by induction)

**Induction Hypothesis**: The first $i$ edges selected by Kruskal’s algorithm are included in some minimal spanning tree $T$.

**Base case**: trivial — the empty set of edges is always in any MST.

**Induction step**: Show that $(i+1)$th edge chosen by Kruskal’s is in the MST $T$ from induction hypothesis, i.e., prove greedy choice property.

- Let $e=(v, w)$ be the edge chosen at $(i+1)$th step of Kruskal’s.
- $T$ is a spanning tree: must include a unique path from $v$ to $w$
- At least one edge $e'$ on this path is not in $X$, the set of edges chosen in the first $i$ steps by Kruskal’s. (Otherwise, $v$ and $w$ will already be connected in $X$ and so $e$ won’t be chosen by Kruskal’s.)
- Since neither $e$ nor $e'$ are in $X$, and Kruskal’s chose $e$, $w(e') \geq w(e)$.
- Replace $e'$ by $e$ in $T$ to get another spanning tree $T'$. Either $w(T') < w(T)$, a contradiction to the assumption $T$ is minimal; or $w(T') = w(T)$, and we have another MST $T'$ consistent with $X \cup \{e\}$. In both cases, we have completed the induction step.
Kruskal’s: Runtime complexity

\[ \text{MST}(V, E, w) \]

\[ X = \phi \]
\[ Q = \text{priorityQueue}(E, w) \quad // \text{from min to max weight} \]
while \( Q \) is nonempty
\[ e = \text{deleteMin}(Q) \]
\[ \text{if } e \text{ connects two disconnected components in } (V, X) \]
\[ X = X \cup \{ e \} \]

- Priority queue: \( O(\log |E|) = O(\log V) \) per operation
- Connectivity test: \( O(\log V) \) per check using a disjoint set data structure
Thus, for \( |E| \) iterations, we have a runtime of \( O(|E| \log |V|) \)

MST: Applications

Network design: Communication networks, transportation networks, electrical grid, oil/water pipelines, ...

Clustering: Application of minimum spanning forest (stop when \( |X| = |V| - k \) to get \( k \) clusters

Broadcasting: Spanning tree protocol in Ethernets

Shortest Paths

Input: A directed graph \( G = (V, E) \), a cost function \( l : E \to \mathbb{R} \)
assigning non-negative costs, source and destination vertices \( s \) and \( t \)

Output: The shortest cost path from \( s \) to \( t \) in \( G \).

Note:
- Single source shortest paths: find shortest paths from \( s \) to all every vertex. Can be solved using the same algorithm, with the same complexity!
- This algorithm constructs a spanning tree called shortest path tree (SPT)

Applications: Routing protocols (OSPF, BGP, RIP, ...), Map routing (flights, cars, mass transit), ...

Dijkstra’s Algorithm: Outline

Base case: Start with \( \text{explored} = \{ s \} \)

Inductive step:
- Optimal substructure: After having computed the shortest path to all vertices in \( \text{explored} \),
- Greedy choice: extend \( \text{explored} \) with a \( v \) that can be reached using one edge \( e \) from some \( u \in \text{explored} \) such that \( \text{dist}(u) + l(e) \) is minimized

Finish: when \( \text{explored} = V \)
Dijkstra’s: High-level intuition

Blue-colored region represents explored, i.e., we have already computed shortest paths to these vertices.

Dijkstra’s Algorithm

\[\text{ShortestPathTree}(V, E, l, s)\]

\[\text{for } v \text{ in } V \text{ do}\]
\[\quad \text{dist}(v) = \infty, \ prev(v) = \text{nil}\]
\[\text{dist}(s) = 0\]
\[H = \text{priorityQueue}(V, \text{dist})\]

\[\text{while } H \text{ is nonempty}\]
\[\quad v = \text{deleteMin}(H) \ \text{// Note: } \text{explored} = V - H\]
\[\text{for } (v, w) \in E \text{ do}\]
\[\quad \text{if } \text{dist}(w) > \text{dist}(v) + l(v, w)\]
\[\quad \text{dist}(w) = \text{dist}(v) + l(v, w)\]
\[\quad \text{prev}(w) = v\]
\[\quad \text{decreaseKey}(H, w)\]
Dijkstra's Algorithm: Correctness

Base case: Start with explored = ∅, so holds vacuously

Induction hypothesis: Tree $T_i$ constructed so far (after $i$ steps of Dijkstra's) is a subtree of an SPT $T$ (Optimal substructure)

Induction step: By contradiction — similar to MST

Let $V_i = V - H$, and $E_i = \{prev(v) | v \in V_i\}$. Note that $T_i = (V_i, E_i)$

Note that $v \in H$ chosen to be added to explored has the lowest dist in $H$. This means its dist must have been updated previously, and must have prev($v$) set to some $u \in$ explored.

Note $T_{i+1} = (V_i \cup \{v\}, E_i \cup (u, v))$. Need to show $(u, v) \in T$.

Since $T$ is a tree, it must have a unique path $P$ from $s$ to $v$.

$P$ must have an edge $(u', v' \in V_i, v' \in H)$ that bridges $V_i$ and $H$.

If $v' = v$ and $u' = u$ we are done. Otherwise:

- if $v' \neq v$ then note that dist($v'$) ≥ dist($v$) (by how $v$ was selected) and hence the so-called shortest path in $T$ to $v$ is longer than that in $T_{i+1}$ — a contradiction. (Assuming $l(x,y) > 0$ for all $x,y \in V$.)
- if $u' \neq u$, then there is still a contradiction if dist($u'$) + $l(u',v)$ > dist($u$) + $l(u,v)$. Otherwise, the two sides should be equal, in which case we can obtain another SPT $T'$ from $T$ by replacing $(u',v)$ by $(u,v)$. This completes the induction step, as we have constructed an SPT consistent with $T_{i+1}$

Dijkstra's Algorithm: Runtime

\[
\text{while } H \text{ is nonempty} \\
\quad v = \text{deleteMin}(H) \\
\quad \text{for } \langle v, w \rangle \in E \text{ do} \\
\quad \quad \text{if } \text{dist}(w) > \text{dist}(v) + l(\langle v, w \rangle) \\
\quad \quad \quad \text{dist}(w) = \text{dist}(v) + l(\langle v, w \rangle) \\
\quad \quad \quad \text{prev}(w) = v \\
\quad \quad \text{decreaseKey}(H, w) \\
\]

- $O(|V|) \text{ iterations of deleteMin: } O(|V| \log |V|)$
- Inner loop executes $O(|E|)$ times, each iteration takes $O(\log V)$ time
- So, total time is $O((|E| + |V|) \log |V|)$

Information Theory and Coding

Information content

For an event $e$ that occurs with probability $p$, its information content is given by $I(e) = - \log p$

- “surprise factor” — low probability event conveys more information; an event that is almost always likely ($p \approx 1$) conveys no information.
- Information content adds up: for two events $e_1$ and $e_2$, their combined information content is $-(\log p_1 + \log p_2)$
Information theory: Entropy

Information entropy
For a discrete random variable $X$ that can take a value $x_i$ with probability $p_i$, its entropy is defined as the expectation ("weighted average") over the information content of $x_i$:

$$H(X) = -\sum_{i=1}^{n} p_i \log p_i$$

- Entropy is a measure of uncertainty
- Plays a fundamental role in many areas, including coding theory and machine learning.

Variable-length encoding
Let $\Sigma = \{A, B, C, D\}$ with probabilities 0.55, 0.02, 0.15, 0.28.
- If we use a fixed-length code, each character will use 2-bits.
- Alternatively, use a variable length code
  - Let us use as many bits as the information content of a character
    - $A$ uses 1 bit, $B$ uses 6 bits, $C$ uses 3 bits, and $D$ uses 2 bits.
    - You get an average saving of 15%
      $$0.55 \cdot 1 + 0.02 \cdot 6 + 0.15 \cdot 3 + 0.28 \cdot 2 = 1.68 \text{ bits}$$
    - Lower bound (entropy)
      $$-(0.5 \log_2 0.5 + 0.02 \log_2 0.02 + 0.14 \log_2 0.14 + 0.27 \log_2 0.27) = 1.51 \text{ bits}$$

Optimal code length
Shannon's source coding theorem
A random variable $X$ denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$
- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits
- The first part of this theorem sets a lower bound, regardless of how clever the encoding is.
- Surprisingly simple proof for such a fundamental theorem! (See Wikipedia.)
- Huffman coding: an algorithm that achieves this bound

Variable-length encoding
Let $\Sigma = \{A, B, C, D\}$ with probabilities 0.55, 0.02, 0.15, 0.28.
- Let us try fixing the codes, not just their lengths:
  - $A = 0$, $D = 11$, $C = 101$, $B = 100$.
  - Note: enough to assign 3 bits to $B$, not 6. So, average coding size reduces to 1.62.

Prefix encoding
- No code is a prefix of another.
- Necessary property to enable decoding.
- Every such encoding can be represented using a full binary tree (either 0 or 2 children for every node)
Huffman encoding

- Build the prefix tree bottom-up
- Start with a node whose children are codewords $c_1$ and $c_2$ that occur least often
- Remove $c_1$ and $c_2$ from alphabet, replace with $c'$ that occurs with frequency $f_1 + f_2$
- Recurse

How to make this algorithm fast?
- What is its complexity?

Huffman encoding: Optimality

- Crux of the proof: *Greedy choice property*
- Familiar exchange argument
  - Suppose the optimal prefix tree does not use longest path for two least frequent codewords $c_1$ and $c_2$
  - Show that by exchanging $c_1$ with the codeword using the longest path in the optimal tree, you can reduce the cost of the “optimal code” — a contradiction
  - Same argument holds for $c_2$

Huffman Coding: Applications

- Document compression
- Signal encoding
- As part of other compression algorithms (MP3, gzip, PKZIP, JPEG, ...)

Uses about 650 bits, vs 850 for fixed-length (5-bit) code.
Lossless Compression

- How much compression can we get using Huffman?
  - It depends on what we mean by a codeword!
    - If they are English characters, effect is relatively small
    - If they are English words, or better, sentences, then much higher compression is possible
- To use words/sentences as codewords, we probably need to construct document-specific codebook
  - Larger alphabet size implies larger codebooks!
  - Need to consider the combined size of codebook plus the encoded document
- Can the codebook be constructed on-the-fly?
  - Lempel-Ziv compression algorithms (gzip)

gzip Algorithm [Lempel-Ziv 1977]

Key Idea: Use preceding $W$-bytes as the codebook (“sliding window”, up to 32KB in gzip)

Encoding:
- Strings previously seen in the window are replaced by the pair $(offset, length)$
  - Need to find the longest match for the current string
  - Matches should have a minimum length, or else they will be emitted as literals
  - Encode offset and length using Huffman encoding

Decoding: Interpret $(offset, length)$ using the same window of $W$-bytes of preceding text. (Much faster than encoding.)

Greedy Algorithms: Summary

- One of the strategies used to solve optimization problems
- Frequently, locally optimal choices are NOT globally optimal, so use with a great deal of care.
  - *Always need to prove optimality*. Proof typically relies on greedy choice property, usually established by an “exchange” argument, and *optimal substructure*.
- Examples
  - MST and clustering
  - Shortest path
  - Huffman encoding