Study Guide: Greedy Algorithms

Drawing on CLRS and Video Lectures

October 13, 2025

1 I. Foundations and Definition

A **Greedy Algorithm** is a technique for solving optimization problems that involves going through a sequence of steps, making a set of choices at each step.

- **Core Principle:** The algorithm always makes the choice that looks best at the moment.
- **Goal:** It makes a **locally optimal choice** in the hope that this choice will lead to a **globally optimal solution**.
- **Limitation:** The greedy strategy typically ignores the effects of the future when making a choice.

2 II. Comparative Analysis: Greedy vs. Dynamic Programming

Both Greedy Algorithms and Dynamic Programming (DP) rely on the property of **Optimal Substructure**. However, they differ in how they make choices.

Greedy vs. Dynamic Programming

Feature	**Greedy Algorithms**	**Dynamic Programming (DP)**
Choice Mechanism	Makes the choice that seems best *at the moment* (locally optimal).	The choice usually depends on the solutions to subproblems.
Processing Direction	Usually progresses **top-down**, making one greedy choice after another, reducing the problem size.	Typically solved in a **bottom-up** manner, progressing from smaller to larger subproblems.
Choice Dependency	Choice cannot depend on any future choices or solutions to subproblems.	Choice depends on subproblem solutions, requiring those solutions to be known first (unless memoizing is used).

Table 1: Comparison of Greedy Algorithms and Dynamic Programming.

Knapsack Problem Contrast (CLRS Example)

The distinction is highlighted by variations of the Knapsack Problem:

- **Fractional Knapsack Problem:** The thief can take fractions of items. This problem **has the greedy-choice property**. The optimal strategy is to sort items by value per pound and take as much as possible of the highest-value items first.
- **0-1 Knapsack Problem:** The thief must make a binary choice (take or leave). The standard greedy strategy (based on value per pound) **does not yield an optimal solution**. This problem exhibits optimal substructure and overlapping subproblems, making it suitable for a DP solution (running time $O(nW)$).

3 III. Core Properties of Greedy Strategy

For a problem to be solvable by a greedy algorithm, it must typically exhibit two essential properties:

3.1 1. Optimal Substructure

The optimal solution to the problem incorporates the optimal solution to subproblems.

- **MST Example:** The Minimum Spanning Tree (MST) exhibits optimal substructure. If an edge (u, v) is removed from an MST T, T is partitioned into two subtrees, T_1 and T_2. The subtree T_1 is itself an MST of the subgraph induced by the vertices of T_1.
- **Proof Technique:** This is proven using the **cut-and-paste argument**, where assuming a suboptimal solution for the subproblem leads to a contradiction regarding the overall optimality of the original solution.
- MST also exhibits **overlapping subproblems**. While this suggests DP, MST also has the stronger greedy choice property.

3.2 2. Greedy Choice Property

You can assemble a globally optimal solution by making locally optimal (greedy) choices. This is the central heuristic distinguishing greedy algorithms.

The Fundamental MST Theorem (Greedy Choice)

Theorem: Let T be the MST of $G = (V, E)$, and let $A \subseteq V$. Suppose $(u, v) \in E$ is the **least-weight edge connecting A to $V \setminus A$** (a cut). Then, (u, v) must be included in T.

- **Significance:** This theorem provides a powerful tool, guaranteeing that the cheapest edge crossing any cut is always a "safe edge" to include in the MST.
- **Proof:** The proof also uses a cut-and-paste argument. If (u, v) were not in T, the unique path between u and v in T must contain some edge e' that also crosses the cut. Since (u, v) is the least-weight crossing edge, $w(u, v) \leq w(e')$. By swapping (u, v) for e' (cutting e' and pasting (u, v)), a new spanning tree T' is formed that is either equally or lighter in weight than T, contradicting the optimality of T if $w(u, v) < w(e')$.

4 IV. Greedy Algorithms: Examples

4.1 1. Minimum Spanning Trees (MST)

The MST problem requires finding a spanning tree T (a connected, acyclic subgraph that includes all vertices) of minimum total weight in a connected, undirected graph $G = (V, E)$.

Prim's Algorithm (Greedy Approach 1)

Prim's algorithm builds the MST by incrementally growing a single tree from an arbitrary starting vertex s.

- **Idea:** Maintain a set A (or S) of vertices already included in the MST. At each step, choose the minimum-weight edge connecting a vertex in A to a vertex outside A (in $V \setminus A$).
- **Mechanism:** Uses a **min-priority queue (Q)** keyed by the weight of the least-weight edge connecting that vertex to A.

```
while Q is not empty:
    u <- EXTRACT-MIN(Q) // u is added to A
    for each neighbor v of u:
        if v in Q and w(u, v) < key[v]:
            key[v] <- w(u, v) // DECREASE-KEY operation
            parent[v] <- u
```

- **Correctness:** Verified using the MST Greedy Choice Theorem.
- **Running Time:** Prim's algorithm runs in time $O(V \cdot T_{\text{Extract-Min}} + E \cdot T_{\text{Decrease-Key}})$.
  - Using a **Fibonacci heap**, the worst-case running time is $O(E + V \lg V)$.

Kruskal's Algorithm (Greedy Approach 2)

Kruskal's algorithm builds a forest of trees, iteratively merging components using the cheapest available edges.

- **Idea:** Select the lowest-weight edge that connects two previously unconnected components (avoiding cycles).
- **Mechanism:**
  1. Initialize $T = \emptyset$ and use `Make-Set(v)` for all vertices.
  2. Sort all edges E by weight in increasing order.
  3. Iterate through sorted edges $e = (u, v)$: if `Find-Set`$(u) \neq$ `Find-Set`(v), add e to T and merge their sets using `Union`(u, v).
- **Data Structure:** Requires the **Disjoint-Set (Union-Find)** data structure to efficiently track connected components.
- **Correctness:** An edge e is added only if it is the minimum-weight edge crossing the cut defined by the two connected components it joins, applying the Greedy Choice Property.
- **Running Time:** Dominated by sorting, $O(E \lg E)$. The total time is $O(E \lg E + E\alpha(V))$, where $\alpha(V)$ is the inverse Ackermann function. If weights are small integers, specialized sorting can make the time nearly linear $O(E + E\alpha(V))$.

4.2 2. Activity Selection Problem (CLRS Chapter Example)

Goal: Schedule a maximum-size subset of mutually compatible activities that require exclusive use of a common resource.

- **Optimal Substructure:** If a maximum compatible set A_{ij} includes activity a_k, it includes optimal solutions to subproblems before a_k finishes (S_{ik}) and after a_k finishes (S_{kj}).
- **Greedy Choice:** Choose the activity that leaves the resource available for as many other activities as possible. This is achieved by selecting the activity with the **earliest finish time**.
- **Algorithm (`GREEDY-ACTIVITY-SELECTOR`):** Assumes activities are sorted by finish time. It iteratively selects the first activity a_k to finish, and then finds the next activity a_m whose start time s_m is not earlier than the finish time f_k of the selected activity.
- **Time Complexity:** If the activities are already sorted, the iterative greedy algorithm runs in $\Theta(n)$ time.

4.3 3. Huffman Codes (CLRS Chapter Example)

Huffman codes create optimal prefix-free binary codes used for data compression by assigning shorter codewords to frequent characters and longer ones to infrequent characters.

- **Greedy Choice:** The tree is built bottom-up by repeatedly performing a merger. The algorithm chooses to merge the two characters/nodes having the **lowest frequencies**. This choice incurs the least cost at the moment, defined as the sum of the frequencies of the two merged items.
- **Mechanism:** Uses a **min-priority queue** (Q) keyed on frequency. It extracts the two nodes of lowest frequency (x and y) and replaces them with a new internal node z whose frequency is $x.freq + y.freq$.
- **Time Complexity:** The total running time is $O(n \lg n)$, assuming the min-priority queue is implemented as a binary min-heap.

4.4 4. Offline Caching (CLRS Chapter Example)

The problem is to minimize the number of cache misses when the entire sequence of future memory requests is known in advance (offline version).

- **Optimal Substructure:** The problem exhibits optimal substructure. An optimal solution S to a subproblem (C, i) incorporates an optimal solution S' to the resulting subproblem $(C', i+1)$.
- **Greedy Choice Strategy:** The **Furthest-in-Future** strategy.
- **Mechanism:** When the cache is full and a miss occurs, evict the block currently in the cache whose next access in the request sequence occurs **furthest in the future**. If a block will never be referenced again, it is chosen for eviction.
- **Optimality:** The furthest-in-future strategy is optimal, proven by showing that the problem has both optimal substructure and the greedy-choice property.

5 Greedy Algorithms Podcast

For a deeper dive into greedy algorithms, you can listen to this podcast covering Greedy Algorithms and Minimum Spanning Trees, including Prim's and Kruskal's algorithms. It was created by Google NotebookLM using the sources in the reference list and is available only in Spanish.

6 Resource Materials

1. **PDF Documents / Book Chapters**

   - **Book Chapter[2]:** Chapter 15 Greedy Algorithms
     - *Context:* Covers general greedy strategy, optimal substructure, greedy-choice property, Activity-Selection Problem, Huffman codes, and Offline Caching.
   - **Book Chapter[1]:** Capítulo 8: Problemas de optimización de grafos y algoritmos codiciosos
     - *Context:* Covers greedy optimization algorithms about graphs, in particular Prim and Kruskal Minimum Spanning Trees. Spanish version of the original English work "*Computer Algorithms: Introduction to Design and Analysis*" published by Addison-Wesley Longman, Inc.
   - **Lecture Slides/PDF[4]:** Lecture 16 Greedy Algorithms (and Graphs)
     - *Context:* Focuses on graph representation, Minimum Spanning Trees (MST), Optimal Substructure, Greedy Choice, and Prim's Algorithm. This material is associated with the Fall 2005 course context.
   - **Lecture Slides/PDF[3]:** Lecture 12 Minimum Spanning Tree
     - *Context:* Provides definitions and proofs for Optimal Substructure and Greedy Choice Property for MST, including detailed pseudocode and runtime analysis for Prim's and Kruskal's algorithms.
   - **Written Lecture Notes/PDF:** Design and Analysis of Algorithms
     - *Context:* Associated written material for the 6.046J / 18.410J course, Spring 2015.

2. **YouTube Video Transcripts**

   - **Video Transcript:** Lec 16 — MIT 6.046J / 18.410J Introduction to Algorithms (SMA 5503), Fall 2005
     - *Content:* Detailed coverage of graph definitions, the MST problem, proof of optimal substructure and the greedy choice theorem, and the derivation and analysis of Prim's Algorithm.
   - **Video Transcript:** 12. Greedy Algorithms: Minimum Spanning Tree
     - *Content:* Explores the definition of greedy algorithms, the contrast between greedy and DP, MST problem definition, edge contraction, the optimal substructure claim, the greedy choice property using cuts, and the introduction of Kruskal's Algorithm.

References

[1] Sara Baase and Allen Van Gelder. *Algoritmos Computacionales: Introducción al análisis y diseño*. Pearson Educación de México, S.A. de C.V., Mexico City, Mexico, 3rd edition, 2002.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 4th edition, 2022.

[3] Erik Demaine. Lecture 12: Greedy algorithms: Minimum spanning tree. https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/resources/lecture-12-greedy-algorithms-minimum-spanning-tree/, 2015. Accessed: 2025-10-11.

[4] Charles Leiserson. Lecture 16: Greedy algorithms, minimum spanning trees. https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/resources/lecture-16-greedy-algorithms-minimum-spanning-trees/, 2005. Accessed: 2025-10-11.