Greedy Algorithms Tutorial

Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit. This tutorial covers essential greedy strategies with explanations and C code.

Activity Selection

The Activity Selection Problem involves selecting the maximum number of non-overlapping activities from a set, where each activity has a start and finish time. The greedy approach optimizes this by:

1. **Sorting activities** in ascending order of finish times.
2. **Selecting the first activity** (earliest finish time) as the starting point.
3. **Iterating through remaining activities**, choosing each subsequent activity only if its start time is ≥ the finish time of the last selected activity.
   This strategy ensures maximum utilization of time without conflicts, as sorting by finish time prioritizes activities that free up resources earliest. The algorithm runs in O(nlog⁡n)O(n \log n)O(nlogn) time due to sorting, followed by a linear scan.

Huffman Coding

Huffman Coding is a lossless data compression algorithm that assigns variable-length codes to characters based on their frequencies:

1. **Frequency calculation**: Count occurrences of each character in the input.
2. **Priority queue construction**: Sort characters by frequency (ascending).
3. **Tree building**:
   - Combine the two least frequent nodes into a new internal node.
   - Assign the sum of their frequencies as the new node's frequency.
   - Repeat until all nodes form a single tree.
4. **Code assignment**: Traverse the tree, assigning `0` to left edges and `1` to right edges.
   This creates **prefix codes** (no code is a prefix of another), enabling unambiguous decoding. The algorithm minimizes total bits required, achieving compression proportional to character frequencies.

Fractional Knapsack

The Fractional Knapsack Problem maximizes the value of items in a knapsack with limited capacity, allowing fractional items:

1. **Calculate value-to-weight ratios**: For each item, compute valueweight\frac{\text{value}}{\text{weight}}weightvalue.

2. **Sort items**: Arrange items in descending order of their ratios.
3. **Greedy selection**:
    - Take as much as possible of the highest-ratio item.
    - If capacity remains, move to the next highest-ratio item.
    - For the last item, take a fraction to fill residual capacity.
      Unlike the 0/1 Knapsack, fractional items enable an optimal greedy solution with O(nlog⁡n)O(n \log n)O(nlogn) time complexity (due to sorting).

Greedy Algorithms for Graphs

Prim's Algorithm

Prim's algorithm constructs a **Minimum Spanning Tree (MST)** for a weighted, undirected graph:

1. **Initialize**: Start with an arbitrary vertex.
2. **Greedy edge selection**: At each step, add the smallest-weight edge connecting a vertex in the MST to a vertex outside it.
3. **Repeat** until all vertices are included.
   Prim's uses a priority queue to efficiently select edges, ensuring the MST has the minimum total edge weight. It runs in O(Elog⁡V)O(E \log V)O(ElogV) time with a binary heap.

Kruskal's Algorithm

Kruskal's algorithm also builds an MST but uses a different approach:

1. **Sort edges**: Arrange all edges in ascending order of weight.
2. **Initialize a forest**: Treat each vertex as a separate tree.
3. **Greedy edge addition**:
    - Iterate through sorted edges.
    - Add an edge to the MST only if it connects two disjoint trees (i.e., avoids cycles).
4. **Terminate** when V−1V-1V−1 edges are selected.
   Kruskal's employs a **Union-Find data structure** to manage connectivity, achieving O(Elog⁡E)O(E \log E)O(ElogE) time.

Key Insights

- **Greedy choice property**: Each algorithm makes locally optimal choices (e.g., earliest finish times, smallest edges) that globally optimize the solution.
- **Efficiency**: Sorting steps dominate time complexity, but heap/UFDS optimizations improve scalability.
- **Applicability**: These techniques underpin scheduling, compression, network design, and resource allocation.

1. Activity Selection Problem

Given start and finish times of activities, select the maximum number of activities that don't overlap.

Approach: Sort activities by finish time. Select the first, then choose the next that starts after the previous ends.

C Code:

```c
#include <stdio.h>
#include <stdlib.h>

struct Activity {
    int start, finish;
};

int compare(const void* a, const void* b) {
    return ((struct Activity*)a)->finish - ((struct Activity*)b)->finish;
}

void activitySelection(struct Activity activities[], int n) {
    qsort(activities, n, sizeof(struct Activity), compare);

    printf("Selected Activities:\n");
    int i = 0;
    printf("(%d, %d)\n", activities[i].start, activities[i].finish);

    for (int j = 1; j < n; j++) {
        if (activities[j].start >= activities[i].finish) {
            printf("(%d, %d)\n", activities[j].start, activities[j].finish);
            i = j;
        }
    }
}
```

---

2. Huffman Coding

Huffman coding is a lossless compression algorithm. Build a binary tree from character frequencies to assign variable-length codes.

C Code:

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
#define MAX_SIZE 100

struct Node {
    char data;
    unsigned freq;
    struct Node *left, *right;
};

struct Node* createNode(char data, unsigned freq) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->freq = freq;
    node->left = node->right = NULL;
    return node;
}

struct Node* buildHuffman(char data[], int freq[], int size) {
    while (size > 1) {
        int min1 = 0, min2 = 1;
        if (freq[min1] > freq[min2]) { int t = min1; min1 = min2; min2 = t; }

        for (int i = 2; i < size; i++) {
            if (freq[i] < freq[min1]) { min2 = min1; min1 = i; }
            else if (freq[i] < freq[min2]) min2 = i;
        }

        struct Node* left = createNode(data[min1], freq[min1]);
        struct Node* right = createNode(data[min2], freq[min2]);
        struct Node* merged = createNode('$', freq[min1] + freq[min2]);
        merged->left = left;
        merged->right = right;

        data[min1] = '$';
        freq[min1] = merged->freq;
        data[min2] = data[size-1];
        freq[min2] = freq[size-1];
        size--;
    }
    return createNode(data[0], freq[0]);
}
```

---

3. Fractional Knapsack

Maximize total value in knapsack by taking fractions of items.

Approach: Sort items by value/weight ratio.

C Code:

```c
#include <stdio.h>
#include <stdlib.h>

struct Item {
    int value, weight;
};

int compare(const void* a, const void* b) {
    double r1 = ((double)((struct Item*)a)->value / ((struct Item*)a)->weight);
    double r2 = ((double)((struct Item*)b)->value / ((struct Item*)b)->weight);
    return (r2 - r1) > 0 ? 1 : -1;
}

double fractionalKnapsack(int W, struct Item arr[], int n) {
    qsort(arr, n, sizeof(struct Item), compare);
    double totalValue = 0.0;
    for (int i = 0; i < n && W > 0; i++) {
        if (arr[i].weight <= W) {
            W -= arr[i].weight;
            totalValue += arr[i].value;
        } else {
            totalValue += arr[i].value * ((double)W / arr[i].weight);
            break;
        }
    }
    return totalValue;
}
```

---

4. Greedy for Graphs

a. Prim's Algorithm

Build MST by always adding the smallest edge that connects a visited and unvisited node.

C Code:

```c
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

#define V 5

int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;
```

```
    for (int v = 0; v < V; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}

void primMST(int graph[V][V]) {
    int parent[V], key[V];
    bool mstSet[V];

    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;

        for (int v = 0; v < V; v++) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    for (int i = 1; i < V; i++)
        printf("%d - %d\n", parent[i], i);
}
```

b. Kruskal's Algorithm

Sort all edges and add to MST using Union-Find to avoid cycles.

C Code:

```
#include <stdio.h>
#include <stdlib.h>

struct Edge {
    int src, dest, weight;
};
```

```c
int find(int parent[], int i) {
    if (parent[i] != i)
        parent[i] = find(parent, parent[i]);
    return parent[i];
}

void Union(int parent[], int x, int y) {
    int xroot = find(parent, x);
    int yroot = find(parent, y);
    parent[xroot] = yroot;
}

int cmp(const void* a, const void* b) {
    return ((struct Edge*)a)->weight - ((struct Edge*)b)->weight;
}

void kruskal(struct Edge edges[], int V, int E) {
    qsort(edges, E, sizeof(edges[0]), cmp);
    int parent[V];
    for (int i = 0; i < V; i++)
        parent[i] = i;

    printf("Edges in MST:\n");
    for (int i = 0, e = 0; e < V - 1 && i < E; i++) {
        int x = find(parent, edges[i].src);
        int y = find(parent, edges[i].dest);
        if (x != y) {
            printf("%d - %d\n", edges[i].src, edges[i].dest);
            Union(parent, x, y);
            e++;
        }
    }
}
```

This tutorial introduces classic greedy algorithms and their applications, particularly in optimization and graph theory. Greedy methods are intuitive and often provide efficient solutions to otherwise complex problems.

Activity Selection

https://www.youtube.com/watch?v=V0ZrLuIVzaY

https://www.youtube.com/watch?v=Qz6D7mrxaJM

https://www.youtube.com/watch?v=U4UoR9vq238

https://www.youtube.com/watch?v=6B7qC5SerlE
https://www.youtube.com/watch?v=15o8hF9eQkA

Huffman Coding

https://www.youtube.com/watch?v=iEm1NRyEe5c
https://www.youtube.com/watch?v=21_bJLB7gyU
https://www.youtube.com/watch?v=7qZV5QDWWQw
https://www.youtube.com/watch?v=co4_ahEDCho
https://www.youtube.com/watch?v=NifsPZ5fLlg

Fractional Knapsack

https://www.youtube.com/watch?v=xZfmHVi7FMg
https://www.youtube.com/watch?v=8LusJS5-AGo
https://www.youtube.com/watch?v=F_DDzYnxO14
https://www.youtube.com/watch?v=13m9ZCB8gjw

Greedy for Graphs (Prim's, Kruskal's)

Prim's Algorithm:

https://www.youtube.com/watch?v=cplfcGZmX7I
https://www.youtube.com/watch?v=jsmMtJpPnhU
https://www.youtube.com/watch?v=4ZlRH0eK-qQ
https://www.youtube.com/watch?v=ZtZaR7EcI5Y

Kruskal's Algorithm:

https://www.youtube.com/watch?v=JptKmWQSerU
https://www.youtube.com/watch?v=PPZGNRmHGzs
https://www.youtube.com/watch?v=4ZlRH0eK-qQ